

TRABALHO DE GRADUAÇÃO

**ESTUDO E DESENVOLVIMENTO DE UMA
FERRAMENTA-BASE PARA CRIAÇÃO DE
PROCESSADORES DE APLICAÇÃO ESPECÍFICA**

Moysés Pinheiro Nery

Brasília, dezembro de 2020



**ENGENHARIA
MECATRÔNICA**
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

**ESTUDO E DESENVOLVIMENTO DE UMA
FERRAMENTA-BASE PARA CRIAÇÃO DE
PROCESSADORES DE APLICAÇÃO ESPECÍFICA**

Moysés Pinheiro Nery

*Relatório submetido como requisito parcial de obtenção
de grau de Engenheiro de Controle e Automação*

Banca Examinadora

Prof. Dr. Jones Yudi Mori Alves da Silva, _____
ENM/UnB
Orientador

Prof. Dr. Daniel Mauricio Muñoz Arboleda, _____
FGA/UnB
Examinador interno

Prof. Dr. Renato Coral Sampaio, FGA/UnB _____
Examinador interno

Brasília, dezembro de 2020

FICHA CATALOGRÁFICA

MOYSÉS, PINHEIRO NERY

Estudo e desenvolvimento de uma ferramenta-base para criação de processadores de aplicação específica,

[Distrito Federal] 2020.

51p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2020). Trabalho de Graduação – Universidade de Brasília. Faculdade de Tecnologia.

1. Back-end LLVM

2. Compiladores

3. Processadores de aplicação específica

I. Mecatrônica/FT/UnB

REFERÊNCIA BIBLIOGRÁFICA

NERY, MOYSÉS PINHEIRO, (2020). Estudo e desenvolvimento de uma ferramenta-base para criação de processadores de aplicação específica. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-*n*°0XX, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 51p.

CESSÃO DE DIREITOS

AUTOR: Moysés Pinheiro Nery

TÍTULO DO TRABALHO DE GRADUAÇÃO: Estudo e desenvolvimento de uma ferramenta-base para criação de processadores de aplicação específica.

GRAU: Engenheiro

ANO: 2020

É concedida à Universidade de Brasília permissão para reproduzir cópias deste Trabalho de Graduação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desse Trabalho de Graduação pode ser reproduzida sem autorização por escrito do autor.

Moysés Pinheiro Nery

Grupo de Automação e Controle - Universidade de Brasília.

70910-900 - Brasília - DF - Brasil.

Agradecimentos

Gostaria de agradecer primeiramente a Deus, que é tudo para mim e sem o qual eu não sou nada. Agradeço também a cada um dos professores, monitores, colegas de turmas que me ajudaram a trilhar minha trajetória acadêmica e a enfrentar cada obstáculo que surgiu. Em especial agradeço ao Professor Jones Yudi pela sua grandiosa ajuda e dedicação durante a orientação deste projeto. Agradeço ainda aos amigos e familiares que sempre me incentivaram cada um de sua maneira. Agradeço aos meus pais, Anísio e Rosângela, por todo o apoio e suporte que me condicionaram a chegar até aqui. Por fim, agradeço à Sophia, minha dedicada, atenciosa e conselheira esposa, por ter sido, a cada dia, minha fonte de inspiração e motivação a seguir em frente.

Moysés Pinheiro Nery

RESUMO

Os sistemas embarcados estão difundidos em quase todos os aspectos do cotidiano da atualidade. Esses sistemas encontram-se em smartphones, videogames, câmeras inteligentes, carros elétricos, fábricas automatizadas e diversas outras aplicações. Por conta disso diversos microprocessadores são desenvolvidos a cada dia para desempenharem certo grupo específico de aplicações, de modo que é necessário acompanhar tais arquiteturas de processadores com seus respectivos compiladores. Este trabalho objetiva estudar o desenvolvimento de um compilador realizado para uma arquitetura computacional didática chamada de “Cpu0”, que é um microprocessador RISC de 32 bits, cujo código em Verilog é aberto ao público, onde o framework LLVM foi utilizado para criação de um backend para portar tal arquitetura de um novo compilador. Além disso, busca-se realizar modificações no backend a fim de adequá-lo a possíveis alterações na ISA do Cpu0 para aplicá-lo a um contexto específico. O LLVM é uma estrutura de compilação modular que foi concebida para compilar softwares de forma otimizada, por meio do fornecimento de informações de alto nível às transformações do compilador que levam a tempos de compilação e execução otimizados.

Palavras Chave: LLVM, Backend, processadores, compiladores

ABSTRACT

Embedded systems are widespread in almost all aspects of today's daily life. These systems are found in smartphones, video games, smart cameras, electric cars, automated factories and several other applications. Because of this, several microprocessors are developed every day to perform a specific group of applications, so it is necessary to add to these processor architectures with their respective compilers. This work aims to study the development of a compiler made for a didactic computational architecture called “Cpu0”, which is a 32-bit RISC microprocessor, whose code in Verilog is open to the public, where the LLVM framework was used to create a backend to port such architecture to a new compiler. In addition, modifications are made to the backend in order to adapt it to possible changes in the Cpu0 ISA to apply it to a specific context. The LLVM is a modular compilation structure that was designed to compile software in an optimized way, by providing high-level information to the compiler transformations that lead to optimized compilation and execution times.

Keywords: LLVM, Backend, processors, compilers

SUMÁRIO

1	Introdução.....	1
1.1	CONTEXTUALIZAÇÃO.....	1
1.2	OBJETIVOS DO PROJETO	4
1.3	APRESENTAÇÃO DO MANUSCRITO.....	5
2	A estrutura do LLVM.....	6
2.1	INTRODUÇÃO	6
2.2	LLVM IR: A REPRESENTAÇÃO INTERMEDIÁRIA	6
2.3	COMPILAÇÃO EM TRÊS FASES	7
2.3.1	FRONTENDS.....	8
2.3.2	OTIMIZADOR	9
2.3.3	BACKENDS	9
2.4	GERAÇÃO DO CÓDIGO DE MÁQUINA	9
2.4.1	ETAPAS DA GERAÇÃO DE CÓDIGO	10
2.4.2	TABLEGEN	11
3	O microprocessador Cpu0.....	13
3.1	BANCO DE REGISTRADORES	13
3.2	CONJUNTO DE INSTRUÇÕES.....	14
3.2.1	INSTRUÇÕES TIPO A	15
3.2.2	INSTRUÇÕES TIPO L	15
3.2.3	INSTRUÇÕES TIPO J.....	15
3.2.4	O REGISTRADOR DE STATUS (SW)	17
3.3	CAMINHO DE DADOS	18
4	O Backend LLVM Cpu0	19
4.1	INFORMAÇÕES GERAIS DO BACKEND.....	21
4.2	SELEÇÃO DE INSTRUÇÕES	22
4.2.1	CONSTRUÇÃO DO DAG INICIAL.....	23
4.2.2	OTIMIZAÇÃO DO DAG INICIAL.....	24
4.2.3	LEGALIZAÇÃO DO DAG	25
4.2.4	SELEÇÃO DE INSTRUÇÕES	28
4.3	ESCALONAMENTO DE INSTRUÇÕES	33

4.4	ALOCAÇÃO DOS REGISTRADORES	34
4.5	INSERÇÃO DO PRÓLOGO E EPÍLOGO	37
4.6	EMIÇÃO DO CÓDIGO	37
4.7	INTEGRAÇÃO AO LLVM.....	38
5	Modificações, Testes e Resultados.....	39
5.1	VERIFICAÇÃO DO BACKEND NO SIMULADOR DE VERILOG.....	39
5.2	OMITINDO UMA INSTRUÇÃO DO BACKEND	45
5.3	INSERINDO UMA INSTRUÇÃO NO BACKEND.....	47
6	Conclusões.....	50
6.1	PERSPECTIVAS FUTURAS.....	50
	REFERÊNCIAS BIBLIOGRÁFICAS	51

LISTA DE FIGURAS

1.1	Esquemático do ASIP Designer extraído de [1].	2
1.2	Esquemático da arquitetura do Xtensa LX7, última versão lançada da plataforma Tensilica Xtensa, extraído de [2].	3
1.3	Esquemático do fluxo de desenvolvimento da TCE extraído de [3].	4
2.1	Código C de duas funções que adicionam dois números.	7
2.2	Código LLVM IR gerado a partir do código em C da Figura 2.1	7
2.3	Estrutura de compilação em três fases do LLVM	8
2.4	Passos de geração de código a partir do LLVM [4].	11
3.1	Estrutura do processador Cpu0, extraído de [5]	13
3.2	Formato das instruções do microprocessador Cpu0.	14
3.3	Formato das instruções tipo A	15
3.4	Formato das instruções tipo L	15
3.5	Formato das instruções tipo J	15
3.6	Formato do registrador SW	17
4.1	Hierarquia das principais classes que compõem o backend do Cpu0.	20
4.2	Implementação da classe CpuTargetMachine que integra o backend.	21
4.3	Implementação da classe CpuSubtarget que integra o backend.	22
4.4	Código LLVM IR da função moduloDiferenca utilizada como exemplo durante o processo de geração de código de máquina.	22
4.5	DAG inicial do bloco "inicio" da função moduloDiferenca.	23
4.6	DAG otimizado do bloco "inicio" da função moduloDiferenca.	25
4.7	Convenções de chamada de função adotadas na construção do backend LLVM Cpu0.	26
4.8	Trecho de código da Cpu0TargetLowering que trata a transformação manual de GlobalAdresses.	27
4.9	DAG legalizado do bloco "inicio" da função moduloDiferenca.	28
4.10	Implementação da classe de instruções genérica dentro do escopo da TableGen.	29
4.11	Implementação da classe de formato de instruções do tipo A dentro do escopo da TableGen.	29
4.12	Implementação da classe de formato de instruções do tipo L dentro do escopo da TableGen.	30

4.13 Implementação da classe de formato de instruções do tipo J dentro do escopo da TableGen.....	30
4.14 Formato da instrução ADD.	30
4.15 Formato da instrução ORi.....	31
4.16 Definição da instrução ADD dentro do escopo da TableGen.	31
4.17 Implementação da classe ArithLogicR, classe utilizada para definir as instruções lógicas e aritméticas cujos operandos são somente registradores.	31
4.18 Definição do operando imediato utilizado pela instrução ORi.	32
4.19 Definição da instrução ORi dentro do escopo da TableGen.....	32
4.20 Implementação da classe ArithLogicI, classe utilizada para definir as instruções lógicas e aritméticas cujos operandos são registradores e valores imediatos.	32
4.21 DAG final do bloco "inicio"da função moduloDiferenca.....	33
4.22 Código gerado a partir do bloco "inicio"da função moduloDiferenca após etapa "Escalonamento de Instruções".	34
4.23 Implementação da classe Cpu0Reg utilizada para definir os registradores do Cpu0 dentro do escopo da TableGen.	34
4.24 Definição do banco de registradores do Cpu0 dentro do escopo da TableGen.	35
4.25 Definição das classes de registradores do Cpu0 dentro do escopo da TableGen.	35
4.26 Código gerado a partir do bloco "inicio"da função moduloDiferenca após a etapa "Alocação de registradores".	37
4.27 Código gerado a partir do bloco "inicio"da função moduloDiferenca após a etapa "Inserção do prólogo e epílogo".	37
4.28 Código final em assembly gerado a partir da função moduloDiferenca.	38
4.29 Método de registro do backend Cpu0 ao LLVM.	38
5.1 Código em C da função que soma dois números quaisquer.	39
5.2 Código em LLVM IR da função que soma dois números quaisquer.....	40
5.3 Código em <i>assembly</i> da função que soma dois números quaisquer.....	40
5.4 Resultado obtido da simulação da execução do código da Figura 5.3.....	40
5.5 Código em C da função que testa a estrutura if-else.	41
5.6 Código em LLVM IR da função que testa a estrutura if-else.....	41
5.7 Código em <i>assembly</i> da função que testa a estrutura if-else.....	42
5.8 Resultado obtido da simulação da execução do código da Figura 5.5.....	42
5.9 Código em C que testa a chamada de função.	43
5.10 Código em LLVM IR da função main.	43
5.11 Código em LLVM IR da função funcSoma.	43
5.12 Código em <i>assembly</i> da função main.	44
5.13 Código em <i>assembly</i> da função funcSoma.	44
5.14 Resultado obtido da simulação da execução do código da Figura 5.9 (1 de 2)	44
5.15 Resultado obtido da simulação da execução do código da Figura 5.9 (2 de 2)	45
5.16 Código LLVM IR utilizado para verificar a omissão de instruções de multiplicação. .	45

5.17 Mensagem de erro obtido ao omitir as definições de instruções de multiplicação no backend Cpu0.....	45
5.18 Comandos <i>SetOperationAction</i> para expandir instruções de multiplicação.	46
5.19 Código em <i>assembly</i> da função da Figura 5.16.....	46
5.20 Código em C da função <i>mulsi3</i>	47
5.21 Operação da instrução MAC.....	47
5.22 Implementação da classe MACFormat	48
5.23 Definição da instrução MAC no arquivo Cpu0InstrInfo.	48
5.24 Função MACTest em LLVM IR utilizada para verificar a implementação da instrução MAC.....	48
5.25 Código <i>assembly</i> gerado a partir da Figura 5.24.....	49

LISTA DE TABELAS

3.1	Lista de registradores do Cpu0.....	14
3.2	Conjunto de instruções da Cpu032I.....	16
3.3	Instruções adicionadas no Cpu032II a partir do Cpu032I.....	17

LISTA DE SÍMBOLOS

Siglas

ABI	<i>Application Binary Interface</i>
ASIP	<i>Application-Specific Instruction Set Processor</i>
CISC	<i>Complex Instruction Set Computer</i>
CPU	Unidade Central de Processamento - <i>Central Processing Unit</i>
DAG	<i>Directed Acyclic Graph</i>
FPGA	<i>Field Programmable Gate Array</i>
GCC	<i>Gnu Compiler Collection</i>
IR	<i>Intermediate Representation</i>
ISA	<i>Instruction Set Architecture</i>
LLVM	<i>Low Level Virtual Machine</i>
MVT	<i>Machine Value Type</i>
RISC	<i>Reduced Instruction Set Computer</i>
SSA	<i>Static Single Assignment</i>
SoC	<i>System On a Chip</i>

Capítulo 1

Introdução

1.1 Contextualização

Os grandes avanços tecnológicos da indústria microeletrônica conseguiram, nas últimas décadas, disseminar a utilização de sistemas de computação para aplicações diversas. Temas atuais, como a Internet das Coisas e a Indústria 4.0, realçam a tendência de adoção de sistemas embarcados. O desempenho de um sistema embarcado é determinado por diversos fatores, tais como: velocidade de processamento, consumo de energia, custo, segurança e confiabilidade, entre outros. Os processadores comumente encontrados no mercado são de uso genérico, oferecendo um desempenho médio apenas razoável. Uma forte tendência tecnológica é a adoção de processadores de aplicação específica, ou seja, processadores desenvolvidos com foco em um domínio de aplicação. Um exemplo são os DSPs (Processadores Digitais de Sinais), os quais possuem instruções e blocos de hardware específicos para os algoritmos da área, como as instruções MAC (*Multiply-And-Accumulate*).

Processadores de aplicação específica são, geralmente, baseados em arquiteturas comuns adaptadas para melhoria de métricas de desempenho (em custo, velocidade, consumo de energia, etc.). Então essas adaptações das arquiteturas devem focar em um domínio de aplicação (telecomunicações, processamento de imagens, inteligência artificial, controle, etc), ou mesmo em aplicações específicas (codificador H264, OCR, reconhecimento de impressão digital, controle PID clássico, etc).

Um dos tipos de processadores de aplicação específica que existe é o ASIP (*Application-Specific Instruction Set Processor*), em que existem instruções criadas especialmente para aquele domínio de aplicação, como o caso da instrução MAC, muito utilizada em DSPs. Atualmente, duas ferramentas comerciais dominam o mercado da área:

- ASIP Designer
- Tensilica xTensa

O ASIP Designer da Synopsys é um conjunto de ferramentas utilizadas na criação de pro-

cessadores customizados [1]. Os principais recursos incluem a rápida exploração de escolhas de arquitetura, geração de um kit de desenvolvimento de software baseado em compilador C/C++ eficiente que se adapta automaticamente a cada mudança na arquitetura e geração automática de RTL (*Register-Transfer Level*) sintetizável.

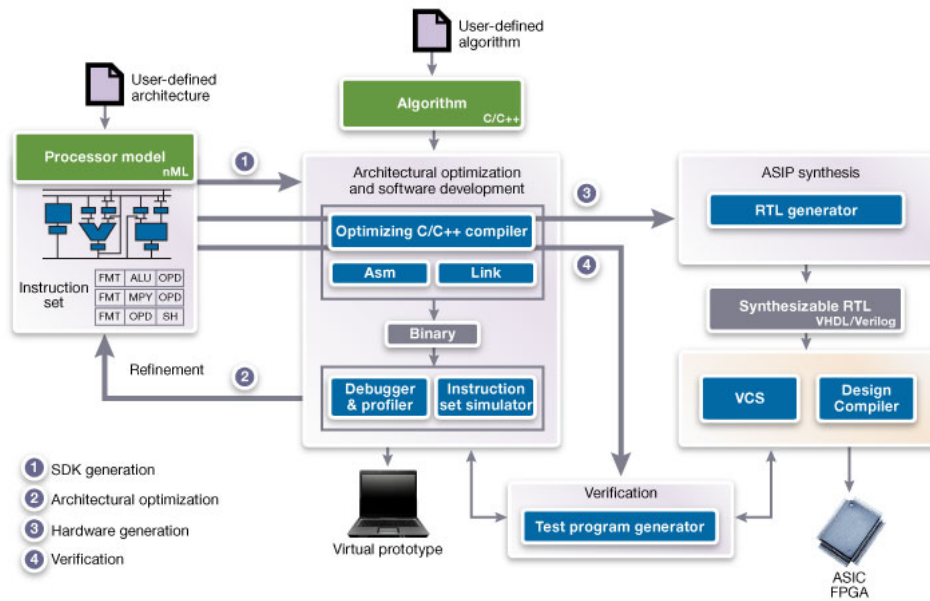


Figura 1.1: Esquemático do ASIP Designer extraído de [1].

Conforme a Figura 1.1, a tecnologia do ASIP Designer oferece suporte à modelagem de arquiteturas de conjunto de instruções ASIP na linguagem de descrição de processador nML. O nML é uma linguagem de definição de alto nível para descrever uma arquitetura de processador e um conjunto de instruções (ISA). Ainda, oferece a tecnologia de compilador em *loop*, permitida pela geração automática de um kit de desenvolvimento de software (SDK) abrangente para cada ASIP modelado em nML, contendo componentes como um compilador otimizador, reconhecido por sua geração de código eficiente e *retargetability* rápida e automática para novas arquiteturas ASIP, um *linker*, que constrói um arquivo executável a partir de arquivos de objeto ELF, um *assembler* e *dissambler*, que traduz o código de máquina da montagem para o formato binário e vice-versa. Ainda, o ASIP Designer vem com uma ampla variedade de exemplos de designs ASIP, permitindo aos desenvolvedores criar rapidamente seu próprio ASIP visando seus requisitos de aplicação específicos.

De forma semelhante, a Tensilica xTensa, da Cadence, é uma ferramenta de geração automática de processadores customizados para aplicações específicas, sendo pioneira neste tipo de mercado [6].

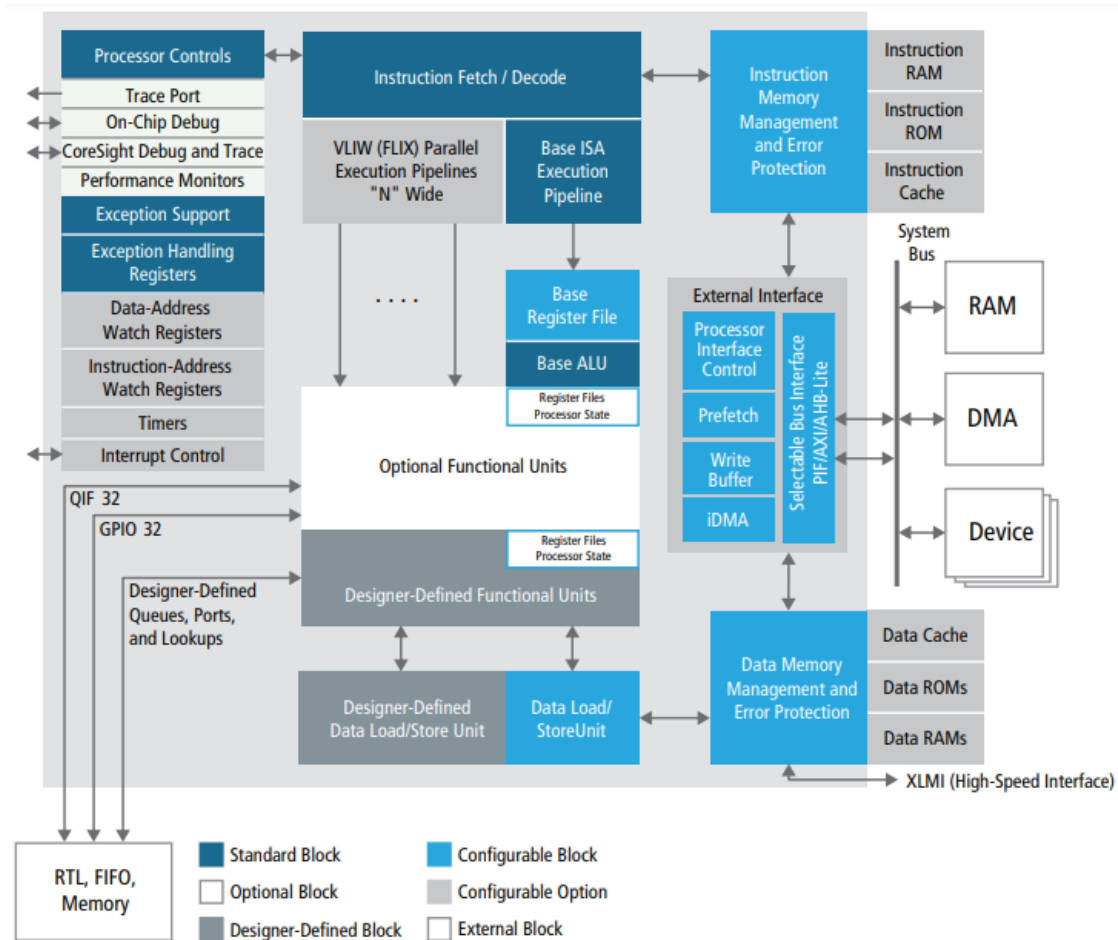


Figura 1.2: Esquemático da arquitetura do Xtensa LX7, última versão lançada da plataforma Tensilica Xtensa, extraído de [2].

Essa ferramenta é bastante flexível, disponibilizando opções de configuração a depender da necessidade do desenvolvedor, conforme Figura 1.2. Além disso, é possível estender a arquitetura base por meio da metodologia *Tensilica Instruction Extension* (TIE). A linguagem TIE pode ser usada para descrever instruções, registradores, unidades funcionais que são adicionados automaticamente para o processador. A linguagem TIE é uma linguagem semelhante a Verilog usado para descrever os mnemônicos de instrução desejados, operandos, codificação e semântica de execução. Arquivos TIE são entradas para o gerador de processadores Xtensa. O gerador automaticamente constrói o processador e a cadeia completa de ferramentas de software que incorpora todas as opções de configuração e novas instruções TIE.

Algumas ferramentas acadêmicas também oferecem processadores customizáveis:

- TCE (*TTA-based Co-Design Environment*)
- LegUp

A TCE vem sendo desenvolvida como *opensource* há mais de 10 anos na Universidade de Tampere, Finlândia. Esse projeto é um conjunto de ferramentas que pode ser usado para projetar

e programar processadores personalizados com base na *Transport Triggered Architecture* (TTA) [3]. O conjunto de ferramentas fornece um fluxo de co-design *retargetable* completo de programas de linguagem de alto nível até o processador RTL sintetizável (backends VHDL e Verilog são suportados) e binários de programa paralelos. Os pontos de personalização do processador incluem o banco de registradores, as unidades funcionais, instruções suportadas e a rede de interconexão.

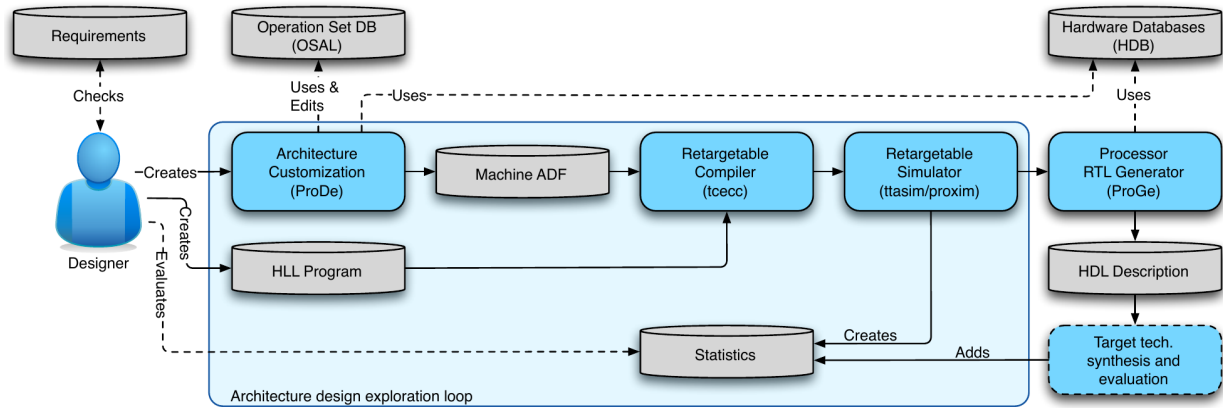


Figura 1.3: Esquemático do fluxo de desenvolvimento da TCE extraído de [3].

A LegUp é uma ferramenta *open source* cuja principal característica é a geração automática de aceleradores em hardware, para um processador baseado em FPGA (*Field Programmable Gate Array*). Essa plataforma permite sintetizar um programa padrão em C em uma arquitetura de processador/acelerador híbrida baseada em FPGA que consiste em um processador se comunicando com aceleradores de hardware personalizados [7]. Dessa maneira, ao compilar uma aplicação em C, a ferramenta analisa a aplicação e gera aceleradores para algumas partes do código. Para isso, ela particiona a aplicação em uma parte para software encapsulada em um programa maior com entradas/saídas; uma parte para hardware, com síntese automática de VHDL/Verilog. Assim, ao final é obtido um processador comum com um acelerador acoplado.

Um dos grandes desafios da criação de processadores de aplicação específica é a criação dos respectivos compiladores, sem os quais dificulta-se muito a implementação de algoritmos complexos. Existem muitos compiladores disponíveis para as mais diversas linguagens de programação, no entanto, a maioria desses compiladores destinam-se a um sistema operacional específico, ou arquitetura preexistente. Não obstante, existem projetos que permitem desenvolver um backend de compilador para novas arquiteturas, onde entre eles encontra-se o LLVM.

1.2 Objetivos do projeto

A proposta deste trabalho é estudar o *framework* LLVM, com ênfase no processo de desenvolvimento de um backend deste *framework* para uma nova arquitetura computacional, tendo como referência o backend desenvolvido para o microprocessador didático Cpu0 [8]. Além disso, objetiva-se realizar modificações no referido compilador de modo a refletir possíveis alterações

realizáveis em processadores de aplicação específica. Os resultados desse projeto servirão de base para o desenvolvimento de uma ferramenta de geração de arquiteturas hardware/software para aplicações específicas.

1.3 Apresentação do manuscrito

Este trabalho está redigido em seis capítulos.

O Capítulo 1 compreende a introdução e apresenta a contextualização, motivações e objetivos a serem alcançados com o trabalho.

No Capítulo 2 é realizada uma descrição da estrutura do LLVM e dos seus principais componentes, com enfoque no processo de geração de código.

No Capítulo 3 encontra-se uma apresentação da organização e arquitetura do processador didático CPU0, bem como a descrição completa do seu conjunto de instruções e caminho de dados.

O Capítulo 4 descreve a implementação do backend LLVM para o processador CPU0, seus principais componentes e forma que os mesmos interagem entre si.

O Capítulo 5 compreende a fase de testes, modificações realizadas e resultados do trabalho.

Por fim, o Capítulo 6 corresponde à conclusão, onde todo o projeto será analisado e possíveis encaminhamentos serão propostos.

Capítulo 2

A estrutura do LLVM

2.1 Introdução

O LLVM é um *framework* de compiladores construído na linguagem C++ e concebido para otimizar a compilação de programas a partir da sua representação intermediária de código, na qual são realizadas diversas transformações que conduzem a uma redução nos tempos de compilação e execução. Começou a ser desenvolvido em 2000 por Chris Lattner na University of Illinois, como tema de sua dissertação de mestrado [9], e passou a ser um *software open source* em 2003, tornando-se um projeto em desenvolvimento contínuo que conta com o apoio de grandes empresas como Intel, Google, Adobe, Apple, entre outras. LLVM é um acrônimo para *Low Level Virtual Machine*, que representava inicialmente sua linguagem intermediária, o que não faz mais sentido visto que o projeto expandiu-se e engloba muitos outros subprojetos [10].

O LLVM vem ganhando grande popularidade entre os desenvolvedores, sendo utilizado em projetos acadêmicos e comerciais. Uma das principais razões para isso é o fato de não ser apenas um compilador, mas sim uma estrutura de compilação baseada em bibliotecas, o que fornece aos seus usuários uma grande flexibilidade, uma vez que seus módulos operam funções específicas e possuem grande independência entre si.

2.2 LLVM IR: a representação intermediária

A característica mais importante da infraestrutura do LLVM é a sua representação intermediária LLVM IR. A LLVM IR é uma linguagem própria, com semântica bem definida.

A LLVM IR é uma representação semelhante a de um conjunto de instruções de um processador RISC (*Reduced-instruction-set Computing*), contendo instruções simples como *add*, *sub*, *compare*, *branch*, entre outras, mas apresentando informações de alto nível que conduzem às otimizações realizadas pelo LLVM. A LLVM IR suporta *labels*; possui um sistema de tipos de dados interno que suporta inteiros, ponto flutuante, booleanos, vetores, *structures*, entre outros; trabalha sobre um número ilimitado de registradores virtuais; pode ser representada em três variações: textual

(extensão .ll) na forma SSA (*Single Static Assignment*), binária (extensão .bc) e uma estrutura de dados “*in-memory*” (do inglês, “em memória”). A descrição completa da linguagem LLVM IR pode ser encontrada no *LLVM Language Manual Reference* [11]. A Figura 2.1 ilustra um código escrito em C e a Figura 2.2 ilustra sua versão correspondente em LLVM IR.

```
unsigned add1(unsigned a, unsigned b){
    return a+b;
}

//perhaps not the most efficient way to add two numbers
unsigned add2(unsigned a, unsigned b){
    if(a == b) return b;
    return add2(a-1, b+1);
}
```

Figura 2.1: Código C de duas funções que adicionam dois números.

```
define i32 @add1(i32 %a, i32 %b) {
    entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
    entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse
recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4
done:
    ret i32 %b
}
```

Figura 2.2: Código LLVM IR gerado a partir do código em C da Figura 2.1

2.3 Compilação em três fases

Assim como a maioria dos compiladores tradicionais, o LLVM adota o projeto de compilador em três fases, conforme Figura 2.3.

Inicialmente, o *frontend* recebe o código fonte como entrada e realiza as análises léxica e sintática, transformando o código na representação intermediária LLVM IR que independe da linguagem do código fonte. Após isso, o otimizador realiza uma série de transformações que visam reduzir o tempo de execução, por exemplo eliminando instruções redundantes. Por fim, o *backend* gera o código de máquina, passando, principalmente, pelas etapas de seleção e escalonamento das instruções, alocação de registradores, otimizações finais e emissão do código de máquina, em formato *assembly* ou binário.

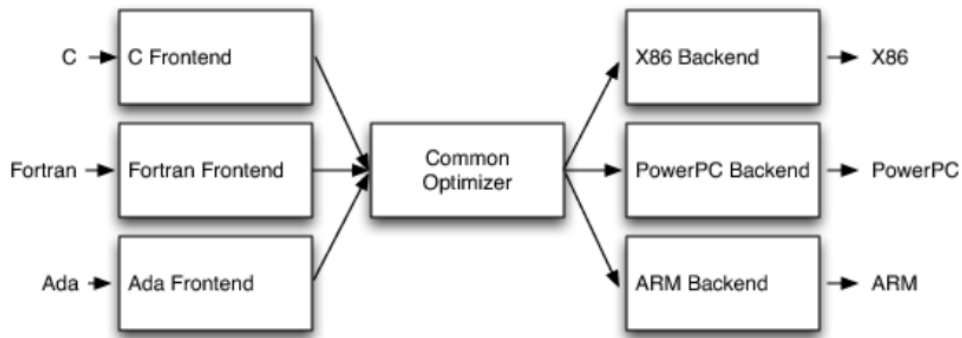


Figura 2.3: Estrutura de compilação em três fases do LLVM

2.3.1 frontends

O LLVM não inclui nenhum frontend para uma linguagem específica. No entanto, sua equipe desenvolveu dois frontends oficiais que são disponibilizados separadamente: GCC-LLVM e Clang. Apesar disso, por ser um framework aberto, há atualmente uma grande comunidade desenvolvendo outros frontends para linguagens como Fortran, Haskell, Ruby, etc. O funcionamento de um frontend LLVM se concentra basicamente em transformar o código de uma linguagem de programação para a linguagem LLVM IR.

2.3.1.1 LLVM-GCC

Como desenvolver um frontend para uma linguagem de programação é uma tarefa bastante trabalhosa e não trivial, a equipe do LLVM decidiu construir o LLVM-GCC, que se baseia no conjunto de frontends preexistentes do GCC 4.2 da Apple. O LLVM-GCC é projetado para transformar o código escrito na linguagem intermediária do GCC, chamada GIMPLE, na linguagem intermediária do LLVM, chamada LLVM IR, de modo que os frontends do GCC funcionem também como frontends do LLVM. A principal desvantagem dessa abordagem é o fato de o frontend do GCC ser lento e consumir bastante memória.

2.3.1.2 Clang

Diferentemente do LLVM-GCC, o Clang é um frontend desenvolvido inteiramente pela equipe do LLVM. Atualmente suporta as linguagens C, C++, Objective-C, Objective-C++, OpenCL, CUDA e RenderScript [12]. Em comparação com o GCC, o Clang apresenta as seguintes vantagens [13][14]:

- Fornece o diagnóstico de erros e cuidados mais claro e descritivo.
- É muito mais rápido e utiliza menos memória.
- Como toda estrutura do LLVM, é modular, extensível, reutilizável.

As desvantagens notáveis em relação ao GCC são o fato de suportar menos linguagens e ser menos popular na comunidade colaborativa.

2.3.2 Otimizador

As otimizações realizadas pelo LLVM independem tanto da linguagem do código fonte quanto da arquitetura para a qual será gerado o código de máquina pelo backend. A partir do código gerado pelo frontend na representação intermediária, são realizadas uma sequência de análises e transformações pelo otimizador do LLVM gerando um código IR mais eficiente e com mesma semântica. Cada uma dessas análises ou transformações são realizadas por algoritmos modulares chamados de *Pass*, de modo a dar liberdade ao projetista para escolher quais operações de otimização serão utilizadas em seu projeto dentre o rol de operações disponibilizadas pelo LLVM.

Para cada uma das formas de apresentação do código em LLVM IR o *framework* fornece ferramentas para realizar as otimizações padrões. A API "*Pass-Manager API*" disponibiliza operações de otimização caso o código LLVM IR esteja na forma "*in-memory*". No caso das formas textual ou binária, a ferramenta *opt* é utilizada para operar as otimizações.

2.3.3 backends

O backend tem a função de transformar o código em LLVM IR para o código de máquina da arquitetura alvo. O LLVM oferece um *framework* de geração de código [4] que disponibiliza algoritmos comuns ao processo de geração de código para todas as arquiteturas, como seleção de instrução, alocação dos registradores, entre outros. A seção a seguir descreve as principais etapas da geração de código e apresenta uma descrição do *framework* de geração de código da LLVM.

2.4 Geração do código de máquina

O gerador de código da LLVM é composto essencialmente por um conjunto de componentes reutilizáveis que permitem a tradução do código em representação intermediária para o código de máquina. Entre eles, pode-se destacar os seguintes [4]:

- Um conjunto de interfaces abstratas que percebem aspectos importantes das características de uma determinada arquitetura alvo;
- Classes utilizadas para representar o processo de geração do código independentemente da arquitetura alvo;
- Classes que modelam o código independentemente da arquitetura alvo.

2.4.1 Etapas da geração de código

O processo de geração de código a partir do *framework* é realizado em sete etapas, conforme a Figura 2.4:

1. Seleção de instrução: primeiramente, o código LLVM IR é transformado de modo a utilizar as instruções da máquina alvo. O processo de seleção de instrução adotado pela LLVM consiste nos seguintes passos:
 - (a) O primeiro passo consiste em transformar o código LLVM em um Grafo Acíclico Direcionado (DAG) chamado de “ilegal” pois utiliza instruções e tipos de dados não suportados pela máquina alvo.
 - (b) Após isso, algumas otimizações são realizadas para simplificar o grafo.
 - (c) Busca-se, então, legalizar o grafo, a fim de substituir as instruções e tipos de dados que não são suportados pela máquina alvo.
 - (d) Algumas outras otimizações são realizadas a fim de eliminar redundâncias geradas após a legalização do grafo.
2. Escalonamento de instruções: a partir do DAG legalizado obtido na etapa anterior, as instruções são reordenadas de modo a satisfazer eventuais restrições da máquina alvo, como por exemplo uma quantidade mínima de registradores disponível em qualquer momento do programa.
3. Otimizações na forma SSA: esta etapa é opcional e consiste em realizar uma série de otimizações no código em SSA, tais como *modulo-scheduling* ou *peephole*.
4. Alocação de registradores: as referências aos registradores virtuais da LLVM IR são redirecionadas aos registradores reais da máquina alvo.
5. Inserção do prólogo e epílogo: uma vez que o código de máquina já foi gerado e o tamanho da pilha necessário já é conhecido, o prólogo e epílogo já podem ser inseridos no código.
6. Otimizações finais: última etapa de otimização, que serve para eliminar eventuais redundâncias que permanecem no código de máquina.
7. Emissão do código: por fim, o código em linguagem de máquina é emitido, completando o processo de compilação.

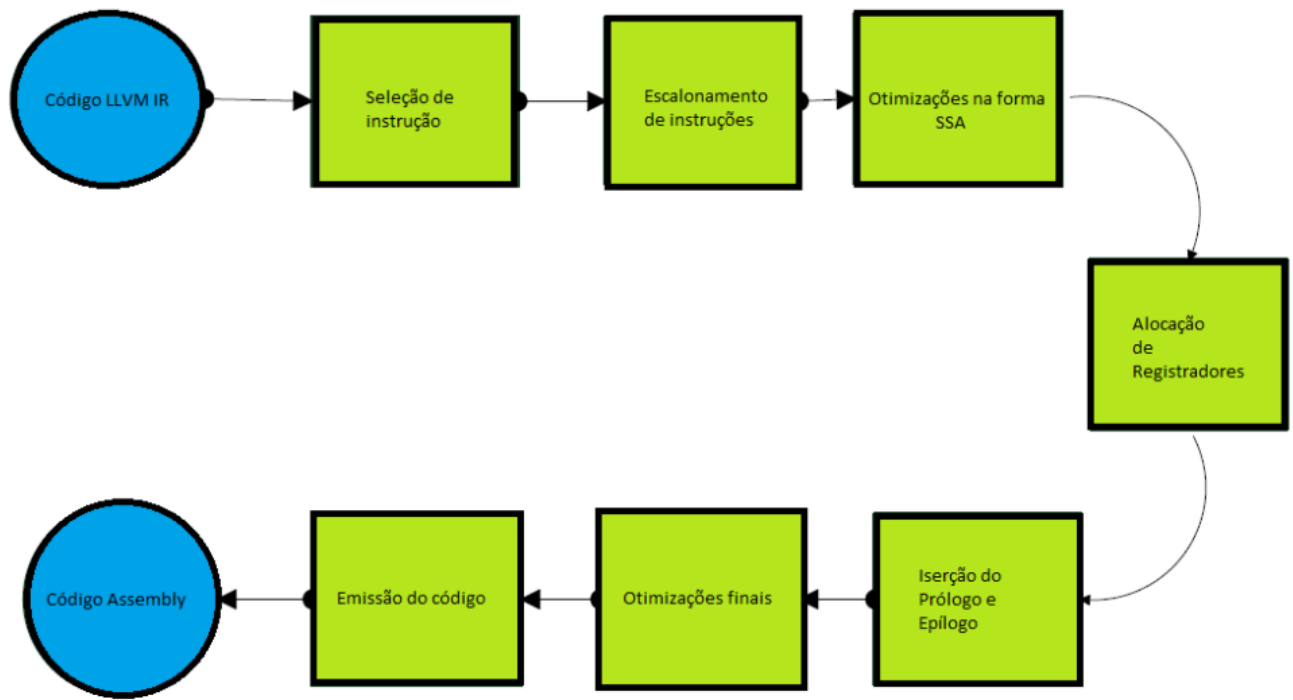


Figura 2.4: Passos de geração de código a partir do LLVM [4]

No desenvolvimento do backend para um novo processador, uma descrição da arquitetura deve ser fornecida ao framework de geração de código para que as etapas acima possam funcionar corretamente. Isso é feito pela implementação de classes que são escritas a partir das classes abstratas fornecidas pelo framework. No entanto, cada uma dessas classes utiliza informações que são comuns entre si, o que acaba conduzindo a redundâncias de código, sujeitando o desenvolvedor a cometer erros facilmente. Diante disso, o LLVM disponibiliza a ferramenta TableGen do LLVM, que busca contornar essa situação.

2.4.2 TableGen

A TableGen [15] é uma ferramenta que é usada para descrever diversos aspectos da arquitetura reduzindo a quantidade de repetição de código. Essa ferramenta gera os códigos em C++ a partir de códigos de descrição que descrevem detalhadamente as características da máquina alvo escritos em sintaxe própria da ferramenta. Componentes como registradores, instruções, convenções de chamadas são descritos a partir da sua estrutura.

A sintaxe utilizada pela TableGen consiste em duas partes principais: classes e definições, onde ambas são chamadas de *records*. As classes são formas abstratas dos *records* e são utilizadas para descrever e construir outros *records*. As definições são instâncias das classes, sendo por isso a forma concreta de um *record*, contendo uma identificação e atribuindo valores aos seus atributos. A TableGen fornece classes abstratas a partir dos quais se pode construir os códigos de descrição da máquina alvo.

É importante salientar que a TableGen não gera todos os componentes necessários, sendo

preciso implementar alguns deles separadamente para se ter o backend completo.

Capítulo 3

O microprocessador Cpu0

O Cpu0 é uma arquitetura de microprocessador de 32 bits desenvolvida pelo professor Chen Zhongcheng, da Golden Gate University [5], para fins didáticos. A Figura 3.1 ilustra os principais componentes da estrutura do Cpu0.

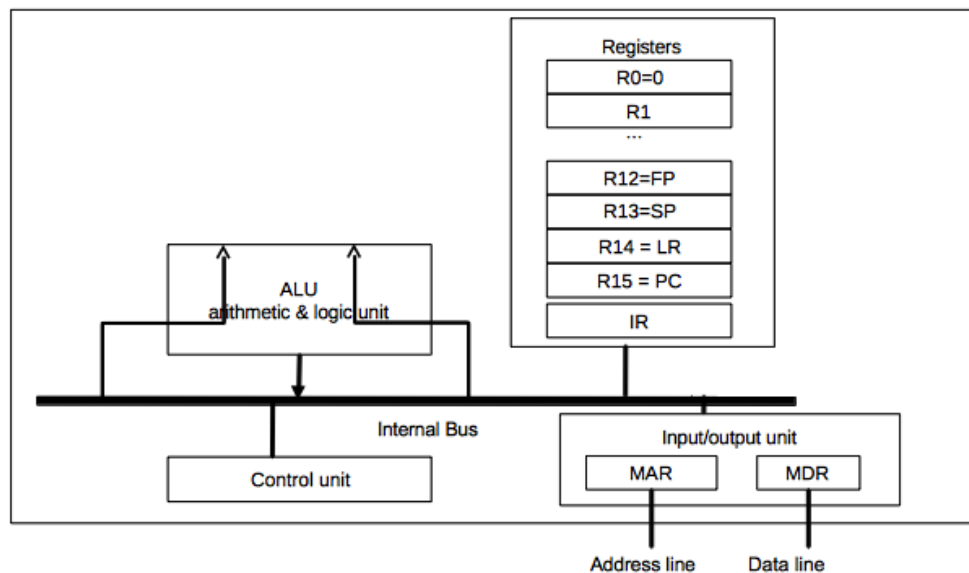


Figura 3.1: Estrutura do processador Cpu0, extraído de [5]

3.1 Banco de registradores

O banco de registradores é constituído por 16 registradores de 32 bits, os quais são enumerados de R0 a R15. O registrador R0 contém sempre o valor zero. Os registradores de R1 a R10 são de uso geral. O registrador R11 corresponde ao *Global Pointer Register* (GP), que aponta para o endereço de memória reservado às variáveis estáticas declaradas no programa em execução. O registrador R12 corresponde ao *Frame Pointer Register* (FP), que aponta para o endereço de memória reservado aos procedimentos. O registrador R13 corresponde ao *Stack Pointer Register*

Registrador	Descrição
R0	Valor fixado em zero
R1-R10	Registradores de uso geral
R11	Global Pointer register (GP)
R12	Frame Pointer register (FP)
R13	Stack Pointer register (SP)
R14	Link Register (LR)
R15	Status Word Register (SW)
IR	Instruction Register
MAR	Memory Address Register (MAR)
MDR	Memory Data Register (MDR)
HI	Guarda os 32 bits mais significativos do resultado de uma multiplicação
LO	Guarda os 32 bits menos significativos do resultado de uma multiplicação

Tabela 3.1: Lista de registradores do Cpu0

(SP), que aponta para o endereço base de memória da pilha. O registrador R14 corresponde ao *Link Register* (LR), que armazena o endereço da próxima instrução a ser executada após uma instrução de salto. Por fim, o registrador R15 corresponde ao *Status Word Register* (SW), que armazena o estado dos campos *Negative* (N), *Zero* (Z), *Carry* (C), *Overflow* (V), *Debug* (D), *Mode* (M), e *Interrupt* (I), que são sinalizadores que servirão para, entre outras coisas, determinar se um desvio será feito ou não.

Além dos registradores mencionados acima, há também outros registradores reservados de uso específico, como o *Program Counter Register* (PC), que contém o endereço de memória da instrução a ser executada, o *Instruction Register* (IR), que armazena a instrução em execução, entre outros. A Tabela 3.1 apresenta a lista completa dos registradores do Cpu0.

3.2 Conjunto de instruções

A arquitetura de conjunto de instruções do Cpu0 foi construída baseada no paradigma RISC e admite três tipos de instruções: instruções de operações lógicas ou aritméticas classificadas como instruções “tipo A”, instruções de acesso à memória que formam a classe de instruções “tipo L” e instruções de desvios do fluxo de execução, chamadas de instruções “tipo J”. As instruções operam em, no máximo, três operandos e possuem o tamanho fixo de 32 bits. O formato dos três tipos de instruções é semelhante, conforme ilustrado na Figura 3.2, o que facilita a decodificação das instruções.



Figura 3.2: Formato das instruções do microprocessador Cpu0

3.2.1 Instruções tipo A

As operações lógicas e aritméticas são realizadas apenas em registradores, ou entre um registrador e um valor imediato. O formato desse tipo de instruções é representado na Figura 3.3.

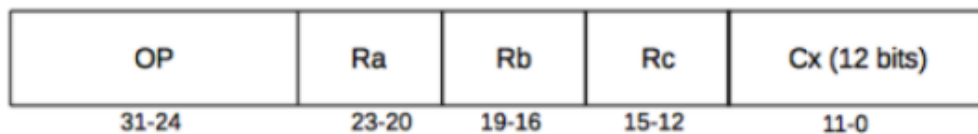


Figura 3.3: Formato das instruções tipo A

O campo Cx é utilizado para guardar valores de deslocamentos de bits ou deslocamento de endereços.

3.2.2 Instruções tipo L

As operações de acesso à memória são efetuadas pelas instruções LOAD e STORE. O formato das instruções tipo L é representado pela Figura 3.4. Nestas instruções, o endereço da memória a ser acessado é obtido pela soma entre Rb e Cx.

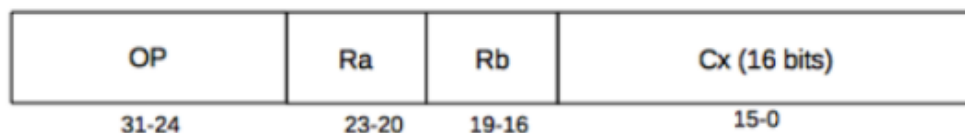


Figura 3.4: Formato das instruções tipo L

3.2.3 Instruções tipo J

Os desvios de fluxo de execução, também chamados de saltos, são tomados ou não a partir do conteúdo do registrador SW (*Status Word*), no qual são guardados resultados de comparações entre dois registradores operandos, conforme descrito anteriormente. O campo Cx contém o deslocamento de endereço do salto em relação ao PC. O formato das instruções tipo J é ilustrado na Figura 3.5.



Figura 3.5: Formato das instruções tipo J

O modelo de arquitetura da Cpu0 admite dois conjuntos de instruções: Cpu032I e Cpu032II. A grande diferença entre esses conjuntos de instruções é que a CPU032II engloba todas as instruções presentes no CPU032I e acrescenta outras. As Tabelas 3.2 e 3.3 retratam as listas das instruções presentes em cada um dos conjuntos de instruções.

Tabela 3.2: Conjunto de instruções da Cpu032I

Formato	Mnemônico	Descrição	Sintaxe
L	NOP	No Operation	
L	LD	Load word	LD Ra, [Rb+Cx]
L	ST	Store word	ST Ra, [Rb+Cx]
L	LB	Load byte	LB Ra, [Rb+Cx]
L	LBu	Load byte unsigned	LBu Ra, [Rb+Cx]
L	SB	Store byte	SB Ra, [Rb+Cx]
L	LH	Load half word	LH Ra, [Rb+Cx]
L	LHu	Load half word unsigned	LHu Ra, [Rb+Cx]
L	SH	Store half word	SH Ra, [Rb+Cx]
L	ADDiu	Add immediate	ADDiu Ra, Rb, Cx
L	ANDi	AND imm	ANDi Ra, Rb, Cx
L	ORi	OR	ORi Ra, Rb, Cx
L	XORi	XOR	XORi Ra, Rb, Cx
L	LUi	Load upper	LUi Ra, Cx
A	CMP	Compare	CMP Ra, Rb
A	ADDu	Add unsigned	ADD Ra, Rb, Rc
A	SUBu	Sub unsigned	SUB Ra, Rb, Rc
A	ADD	Add	ADD Ra, Rb, Rc
A	SUB	Subtract	SUB Ra, Rb, Rc
A	MUL	Multiply	MUL Ra, Rb, Rc
A	AND	Bitwise and	AND Ra, Rb, Rc
A	OR	Bitwise or	OR Ra, Rb, Rc
A	XOR	Bitwise exclusive or	XOR Ra, Rb, Rc
A	ROL	Rotate left	ROL Ra, Rb, Cx
A	ROR	Rotate right	ROR Ra, Rb, Cx
A	SRA	Shift right	SRA Ra, Rb, Cx
A	SHL	Shift left	SHL Ra, Rb, Cx
A	SHR	Shift right	SHR Ra, Rb, Cx
A	SRAV	Shift right	SRAV Ra, Rb, Rc
A	SHLV	Shift left	SHLV Ra, Rb, Rc
A	SHRV	Shift right	SHRV Ra, Rb, Rc
A	ROL	Rotate left	ROL Ra, Rb, Rc
A	ROR	Rotate right	ROR Ra, Rb, Rc
J	JEQ	Jump if equal (==)	JEQ Cx
J	JNE	Jump if not equal (!=)	JNE Cx
J	JLT	Jump if less than (<)	JLT Cx
J	JGT	Jump if greater than (>)	JGT Cx
J	JLE	Jump if less than or equals (<=)	JLE Cx

J	JGE	Jump if greater than or equals (\geq)	JGE Cx
J	JMP	Jump (unconditional)	JMP Cx
J	JALR	Indirect jump	JALR Rb
J	JSUB	Jump to subroutine	JSUB Cx
J	JR/RET	Return from subroutine	JR \$1 or RET LR
A	MULT	Multiply for 64 bits result	MULT Ra, Rb
A	MULTU	MULT for unsigned 64 bits	MULTU Ra, Rb
A	DIV	Divide	DIV Ra, Rb
A	DIVU	Divide unsigned	DIVU Ra, Rb
A	MFHI	Move HI to GPR	MFHI Ra
A	MFLO	Move LO to GPR	MFLO Ra
A	MTHI	Move GPR to HI	MTHI Ra
A	MTLO	Move GPR to LO	MTLO Ra
A	MFC0	Move C0R to GPR	MFC0 Ra, Rb
A	MTC0	Move GPR to C0R	MTC0 Ra, Rb
A	C0MOV	Move C0R to C0R	C0MOV Ra, Rb

Tabela 3.3: Instruções adicionadas no Cpu032II a partir do Cpu032I

Formato	Mnemônico	Descrição	Sintaxe
L	SLTi	Set less Than	SLTi Ra, Rb, Cx
L	SLTiu	SLTi unsigned	SLTiu Ra, Rb, Cx
A	SLT	Set less Than	SLT Ra, Rb, Rc
A	SLTu	SLT unsigned	SLTu Ra, Rb, Rc
L	BEQ	Branch if equal	BEQ Ra, Rb, Cx
L	BNE	Branch if not equal	BNE Ra, Rb, Cx
J	BAL	Branch and link	BAL Cx

3.2.4 O registrador de status (SW)

O registrador de status (SW) da Cpu0 contém o estado dos campos sinalizadores: *Negative* (N), *Zero* (Z), *Carry* (C), *Overflow* (V), *Debug* (D), *Mode* (M) e *Interrupt* (I). Sua configuração é ilustrada na Figura 3.6 a seguir.

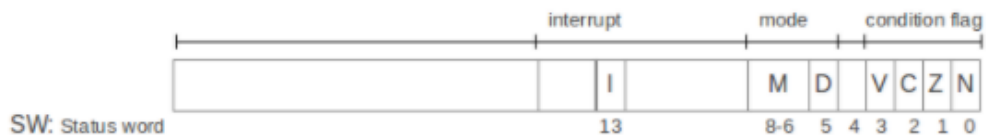


Figura 3.6: Formato do registrador SW

Esses campos são modificados a partir da execução de determinadas instruções. Por exemplo,

ao executar a instrução CMP Ra, Rb, temos que:

- Se $Ra > Rb$, então $N = 0$ e $Z = 0$.
- Se $Ra < Rb$, então $N = 1$ e $Z = 0$.
- Se $Ra = Rb$, então $N = 0$ e $Z = 1$.

São utilizados para, entre outras finalidades, determinar se um salto condicional será tomado ou não, identificar erros e falhas, acionar interrupções, etc.

3.3 Caminho de dados

O Cpu0 é uma implementação pipeline de cinco estágios: *fetch*, *instruction decode*, *execution*, *memory access* e *write-back*.

No estágio *fetch*, a instrução é obtida pelo endereço de memória guardado no PC e é armazenada em IR. Após isso, PC é incrementado em quatro (uma vez que o endereçamento é dado em byte e o tamanho das instruções é de 32 bits.).

No estágio *instruction decode*, a unidade de controle do Cpu0 decodifica a instrução armazenada em IR, envia os valores dos registradores às entradas da ULA e configura o modo de operação da ULA baseado no campo opcode, bem como envia os sinais de controle para os próximos estágios.

No estágio *execute*, a ULA executa a operação determinada e armazena o resultado no registrador de destino, a menos que se trate de uma instrução que não seja lógico-aritmética.

No estágio *memory access*, quando for uma instrução LOAD, o dado da memória é armazenado no registrador MEM/WB. Quando for uma instrução STORE, o dado do registrador de origem é armazenado na memória.

No estágio *write-back*, usado apenas quando se tratar de uma instrução LOAD, o dado é repassado do registrador MEM/WB para o registrador de destino.

Capítulo 4

O Backend LLVM Cpu0

Este Capítulo apresenta o desenvolvimento do backend LLVM para o processador didático Cpu0, o qual foi realizado por Chen Chung-Shu [8]. No desenvolvimento, foi utilizado o *framework* de geração de código do LLVM - também chamado de “gerador de código” (é possível desenvolver um backend LLVM “do zero”, no entanto é um processo bem mais dispendioso [16]). O gerador de código fornece uma hierarquia de classes base a partir das quais são implementadas subclasses que fornecem informações relativas às características e peculiaridades da máquina alvo. O *framework* também disponibiliza um conjunto de algoritmos - que são independentes de qualquer máquina alvo, e por isso podem ser utilizados para construção de geradores de código para uma variedade de processadores - que implementam fases da geração do código, como por exemplo: seleção de instruções, escalonamento de instruções e alocação de registradores, entre outras. Esses algoritmos fazem uso, portanto, das informações específicas da máquina para a qual se desenvolve o backend, de modo que o gerador de código funcione corretamente. A Figura 4.1 ilustra a hierarquia das principais classes implementadas. Em verde encontra-se as classes basilares fornecidas pelo gerador de código. Em amarelo encontra-se as classes implementadas manualmente para o backend Cpu0. Em azul encontram-se as classes geradas automaticamente pela ferramenta TableGen a partir dos arquivos de descrição.

A organização deste Capítulo é composta da seguinte forma: a Seção 4.1 apresenta os primeiros passos na construção do backend; as Seções 4.2 a 4.6 descrevem os trabalhos realizados para implementar as etapas mais relevantes da geração do código de máquina; por fim, a Seção 4.7 apresenta como o backend é integrado à estrutura do LLVM a fim de poder finalmente ser utilizado.

4.1 Informações gerais do backend

O primeiro passo na construção do backend LLVM para uma nova arquitetura é a implementação de uma subclasse da classe basilar *TargetMachine*. Essa classe é utilizada, entre outros fins, para que o gerador de código possa acessar os diversos componentes do backend utilizados no processo de geração de código, como o conjunto de instruções, banco de registradores, *stack frame*, etc. Além disso, ela define algumas características, como: o *layout* dos dados, tamanho do ponteiro, se é adotado *big endian* ou *little endian*. A classe *Cpu0TargetMachine* foi implementada para este fim (Figura 4.2). No caso do backend do *Cpu0*, o desenvolvedor do backend decidiu implementar essa interface no escopo da classe *Cpu0Subtarget*, como pode ser visto na Figura 4.3, onde definiu-se uma série de métodos que acessam os respectivos componentes. Caso algum módulo não faça referência à classe *Subtarget*, a mesma pode ainda ser acessada por meio da *Cpu0TargetMachine* a partir de um *static cast*.

```
Cpu0TargetMachine::Cpu0TargetMachine(const Target &T, const Triple &TT,
                                     StringRef CPU, StringRef FS,
                                     const TargetOptions &Options,
                                     Optional<Reloc::Model> RM,
                                     CodeModel::Model CM, CodeGenOpt::Level OL,
                                     bool isLittle)
: LLVMTargetMachine(T, computeDataLayout(TT, CPU, Options, isLittle), TT,
                   CPU, FS, Options, getEffectiveRelocModel(CM, RM), CM,
                   OL),
  isLittle(isLittle), TLOF(make_unique<Cpu0TargetObjectFile>()),
  ABI(Cpu0ABIInfo::computeTargetABI()),
  DefaultSubtarget(TT, CPU, FS, isLittle, *this) {
}

static std::string computeDataLayout(const Triple &TT,
                                     StringRef CPU,
                                     const TargetOptions &Options,
                                     bool isLittle) {
    std::string Ret = "";
    if (isLittle)
        Ret += "e";
    else
        Ret += "E";
    Ret += "-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64";
    return Ret;
}
```

Figura 4.2: Implementação da classe *CpuTargetMachine* que integra o backend.

```

class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
    virtual void anchor();

public:
    ...
    Cpu0Subtarget &initializeSubtargetDependencies(StringRef CPU, StringRef FS,
                                                    const TargetMachine &TM);
    const SelectionDAGTargetInfo *getSelectionDAGInfo() const override {
        return &TSInfo;
    }
    const Cpu0InstrInfo *getInstrInfo() const override { return InstrInfo.get(); }
    const TargetFrameLowering *getFrameLowering() const override {
        return FrameLowering.get();
    }
    const Cpu0RegisterInfo *getRegisterInfo() const override {
        return &InstrInfo->getRegisterInfo();
    }
    const Cpu0TargetLowering *getTargetLowering() const override {
        return TLInfo.get();
    }
    const InstrItineraryData *getInstrItineraryData() const override {
        return &InstrItins;
    }
};

```

Figura 4.3: Implementação da classe CpuSubtarget que integra o backend.

4.2 Seleção de instruções

A seleção de instruções é a primeira etapa do processo de geração de código. Esta etapa é dividida em algumas fases, conforme visto no Capítulo 2, cujas implementações são demonstradas nas subseções abaixo. A Figura 4.4 retrata uma função em LLVM IR que será usada como exemplo para elucidar as transformações sofridas nos DAGs durante esta etapa. A função recebe dois números inteiros como parâmetros e retorna o módulo da subtração entre esses números.

```

define i32 @moduloDiferenca (i32 %n1, i32 %n2) nounwind {
inicio:
    %tmp = sub nsw i32 %n1, %n2
    %cmp = icmp sge i32 %tmp, 0
    br il %cmp, label %then, label %else
then:
    %result1 = sub nsw i32 %n1, %n2
    ret i32 %result1;
else:
    %result2 = sub nsw i32 %n2, %n1
    ret i32 %result2;
}

```

Figura 4.4: Código LLVM IR da função moduloDiferenca utilizada como exemplo durante o processo de geração de código de máquina.

4.2.1 Construção do DAG inicial

O código de entrada do backend, na representação intermediária LLVM, é, inicialmente, transformado em um grafo acíclico direcionado (DAG). As classes SelectionDAGBuild e TargetLowering do gerador de código são responsáveis pela construção do DAG inicial e, portanto, subclasses delas foram implementadas no backend do Cpu0. Para cada bloco de código LLVM é gerado um DAG (no caso da função da Figura 4.4, é gerado um DAG para o bloco "inicio", "then" e "else") e para cada instrução deste bloco é criado um nó do DAG, que são instâncias da classe SDNode. Cada nó possui um opcode, para identificar a operação, e seus operandos, caso houver. Cada nó define um ou mais valores aos quais são associados seus respectivos MVT (Machine Value Type), que indica o tipo de resultado. Há dois tipos de valores como resultado: os que representam dados propriamente dito (inteiros ou ponto flutuantes) e os que envolvem fluxo de controle, representados por ligações em cadeias que permitem definir a ordem entre os nós de efeitos secundários (como exemplo os loads, stores, calls, returns, etc). Os nós desse tipo recebem um *token chain* como entrada e geram um novo *token chain* como saída. No caso do bloco "inicio" da função da Figura 4.4, o DAG inicial gerado é:

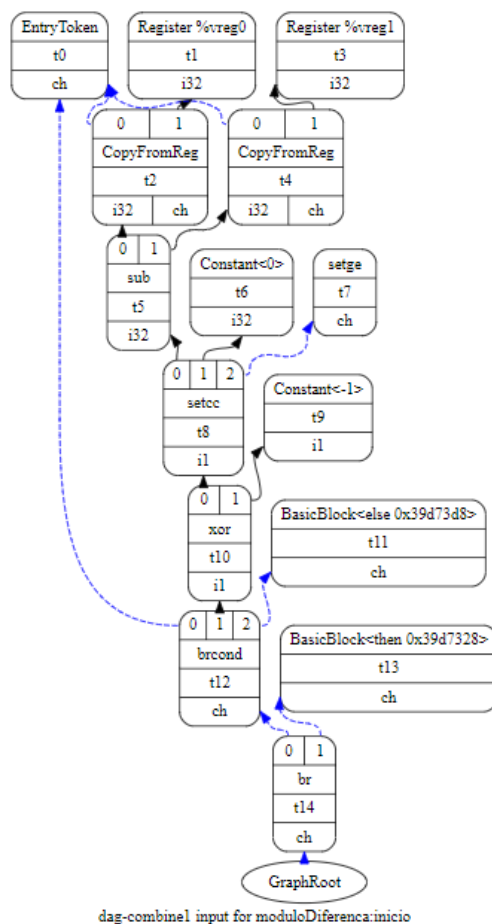


Figura 4.5: DAG inicial do bloco "inicio" da função moduloDiferenca.

Os valores dos parâmetros de entrada são copiados aos registradores t2 e t4. Nota-se que a

instrução **icmp** é transformada em **setcc**, que recebe o resultado da subtração, a constante 0 e a condição de comparação como parâmetros de entrada. A instrução **br** é transformada em **brcond** e recebe como parâmetros de entrada o nó **EntryToken**, o resultado de **setcc** invertido e o endereço de destino do salto caso **setcc** resulte logicamente em falso (ou seja, o endereço do bloco "else"). Por fim, o nó **br** recebe como parâmetro o nó **brcond** e t13 que contém o endereço de destino (bloco "then") caso **brcond** não tenha sido efetivado.

4.2.2 Otimização do DAG inicial

Antes da etapa de legalização do DAG, o gerador de código realiza um passo de otimização no DAG inicial a fim de tornar o código mais limpo, independente da máquina alvo. O DAG da Figura 4.5 é transformado no da Figura 4.6. Percebe-se que o tipo de comparação é alterado (de **setge** para **setlt**) de maneira que não é mais necessário o nó **XOR**.

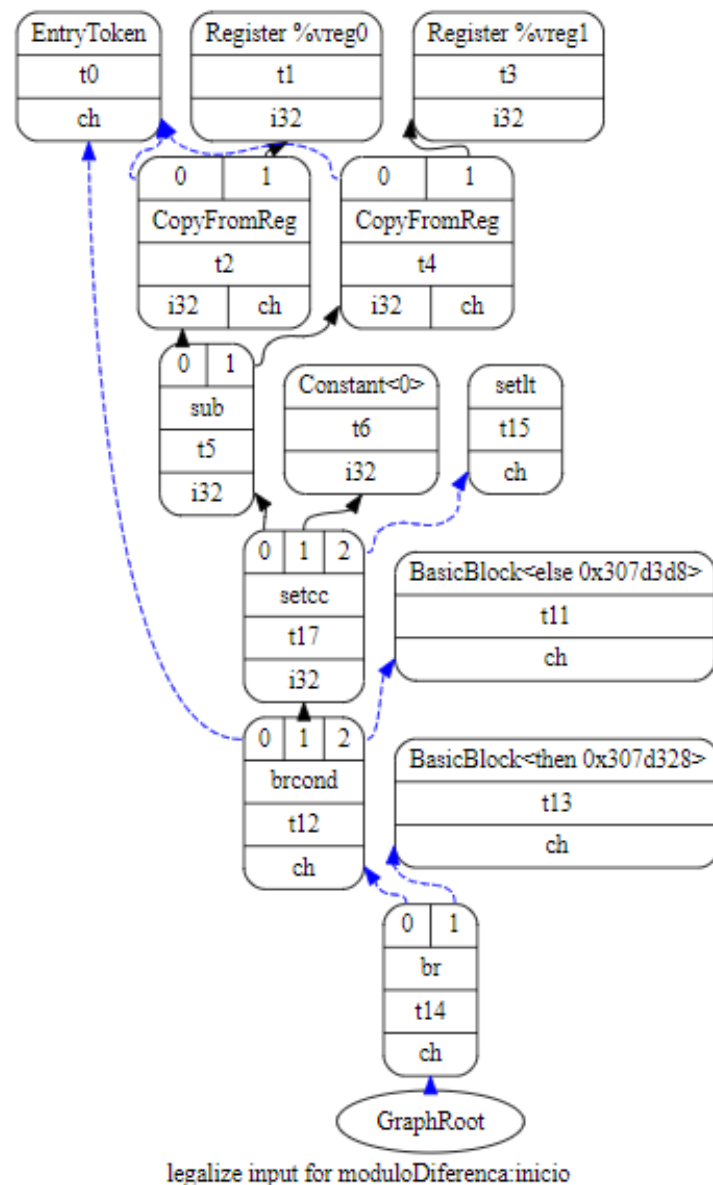


Figura 4.6: DAG otimizado do bloco "inicio" da função moduloDiferenca.

4.2.3 Legalização do DAG

A fase de legalização do DAG é a responsável por transformar o DAG, dando tratamento a todas as instruções e tipos de dados não suportados pela máquina alvo. É necessário, para tanto, fornecer ao gerador de código informações específicas da CPU, entre elas:

- Como expandir ou contrair instruções e/ou operandos a fim de que possam ser utilizadas pela CPU?
- Como tratar as instruções que não podem ser transformadas automaticamente?
- Quais são os tipos de dados suportados para cada instrução e, para cada um deles, a classe de registradores que deve ser utilizada?

- Como tratar as convenções de chamadas de função?

A classe `TargetLowering` é disponibilizada para fornecer grande parte dessas informações ao gerador de código e, por isso, sua subclasse `Cpu0TargetLowering` foi implementada para este fim. As subseções abaixo apresentam suas principais implementações.

4.2.3.1 Convenções de chamada

As convenções de chamada correspondem ao modo como argumentos e valores de retorno são repassados durante as chamadas de funções para uma determinada máquina-alvo. A depender da ABI (*Application Binary Interface*), um conjunto de restrições podem ou não ser exigido para seu correto funcionamento. No caso do `Cpu0`, a única restrição estabelecida é ilustrada na Figura 4.7, definida no arquivo TableGen “`Cpu0CallingConv.td`”. Ela define que para os valores de retorno, quando de tamanho 32 bits, devem ser utilizados os registradores `V0`, `V1`, `A0` e `A1`, nessa ordem de prioridade.

```
def RetCC_Cpu0EABI : CallingConv<[  
  // i32 are returned in registers V0, V1, A0, A1  
  CCIIfType<[i32], CCAssignToReg<[V0, V1, A0, A1]>>  
1>;
```

Figura 4.7: Convenções de chamada de função adotadas na construção do backend LLVM `Cpu0`.

A `Cpu0TargetLowering` é responsável por tratar cada chamada feita a uma função. Ao chamar uma função, entrar em seu escopo e retornar a execução, as seguintes funções são executadas, respectivamente:

`LowerCall()`: Quando uma função é chamada, antes é necessário passar todos os argumentos ao local apropriado. A função `LowerCall()` é a responsável por esta tarefa. Ela calcula, inicialmente, o quanto de memória é preciso e passa esse valor como um argumento da pseudo-instrução `CALLSEQ__START` que insere uma sequência de **stores** para passar os argumentos à pilha. Em seguida, a instrução **call** é então inserida e, por fim, a pseudo-instrução `CALLSEQ__END`, que indica ao *framework* o fim da sequência.

`LowerFormalArguments()`: Essa função é executada para garantir que o escopo da função chamada possa acessar os argumentos, o que significa estarem disponíveis em registradores virtuais. Aqueles que já estão em registradores são simplesmente copiados para os registradores virtuais. Aqueles que estão na pilha são carregados para os registradores virtuais através da inserção de instruções **load**.

`LowerReturn()`: Ao sair do escopo da função chamada, a função `LowerReturn()` é invocada. O valor de retorno, caso exista, é passado a algum registrador físico conforme as convenções de chamada e um nó `RET__FLAG` é inserido no DAG para que, na fase de seleção de instrução, seja substituído pela instrução nativa de retorno **ret**.

4.2.3.2 Transformação manual de instruções ilegais

Geralmente, há instruções e/ou operandos que precisam ser transformadas manualmente devido a certas propriedades e restrições do microprocessador. A implementação dessas transformações se dá tanto em arquivos de descrição TableGen quanto em códigos em C++. Classes customizadas da SDNode são primeiramente definidas em Cpu0InstrInfo.td e as operações são especificadas no construtor da Cpu0TargetLowering. Nesta etapa substituí-se os nós em questão por outros nós intermediários que possam ser utilizados na etapa de seleção de instrução. Entre as transformações mais comuns realizadas nos backends LLVM estão as adequações para instruções de saltos condicionais e de *shift* de bits, pois muitas vezes a arquitetura não possui um registrador com *flags* de comparação nem instruções de *shift* que possam ser transformadas automaticamente a partir das instruções contidas na LLVM IR.

No caso do Cpu0, entre as poucas transformações manuais realizadas tem-se as que dizem respeito a endereços virtuais absolutos, como no caso de *Global Addresses*. Basicamente o endereço é dividido em duas partes - Hi e Lo - e um nó ADD é inserido para somar essas partes (conforme Figura 4.8), possibilitando ao seletor de instruções posteriormente converter essa adição em um endereço de memória utilizado pelas instruções de acesso a memória.

```
SDValue Cpu0TargetLowering::
lowerGlobalTLSAddress(SDValue Op, SelectionDAG &DAG) const
{
    ...
    SDValue TGAHi = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                                Cpu0II::MO_DTP_HI);
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, PtrVT, TGAHi);
    SDValue TGALo = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                                Cpu0II::MO_DTP_LO);
    SDValue Lo = DAG.getNode(Cpu0ISD::Lo, DL, PtrVT, TGALo);
    SDValue Add = DAG.getNode(ISD::ADD, DL, PtrVT, Hi, Lo);
    return DAG.getNode(ISD::ADD, DL, PtrVT, Add, Lo);
    ...
}
```

Figura 4.8: Trecho de código da Cpu0TargetLowering que trata a transformação manual de GlobalAddresses.

O DAG ilegal da Figura 4.6 é então transformado no DAG legalizado da Figura 4.9.

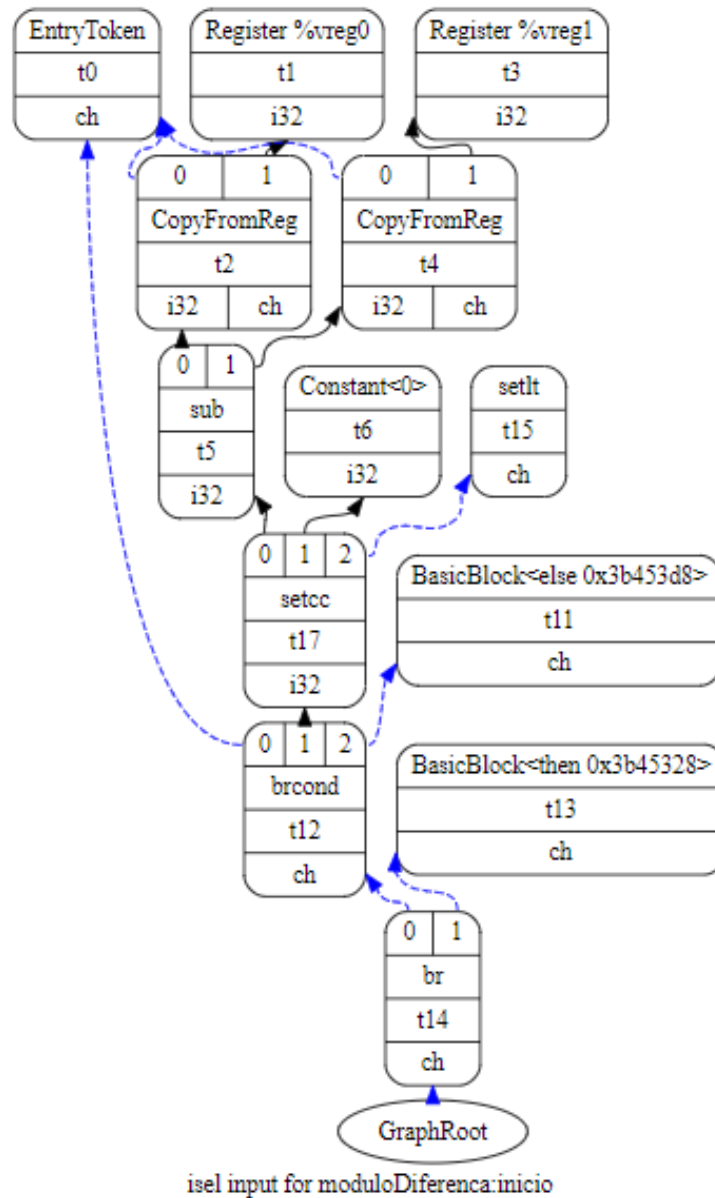


Figura 4.9: DAG legalizado do bloco "inicio" da função moduloDiferenca.

4.2.4 Seleção de instruções

Uma vez que o DAG esteja legalizado, ele deve passar por uma última transformação, onde seus nós são transformados em instruções que são suportadas pelo Cpu0 de forma nativa. Para isso, o gerador de código necessita de informações detalhadas a respeito do conjunto de instruções do Cpu0. Conforme visto no Capítulo 2, a TableGen exerce esse papel. Os arquivos Cpu0InstrFormats.td e Cpu0InstrInfo.td contêm a descrição dos formatos das instruções e suas definições de fato, respectivamente.

4.2.4.1 Descrição do formato de instruções

As instruções devem ser especificadas a partir da classe `Instruction`. A classe `Cpu0Inst` foi implementada como uma subclasse daquela e é utilizada na implementação das classes que especificam os formatos de instruções do `Cpu0`. Conforme a Figura 4.10, o *opcode* é definido como os oito bits mais significativos da instrução, uma vez que seu tamanho e posição são os mesmos para todos os formatos de instruções, conforme visto no Capítulo 3. Além disso, é possível definir quantos operandos a instrução tem, seus respectivos tamanhos e mapeamento no código da instrução.

```
// Generic Cpu0 Format
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
            InstrItinClass itin, Format f>: Instruction
{
    field bits<32> Inst;
    Format Form = f;
    let Namespace = "Cpu0";
    let Size = 4;
    bits<8> Opcode = 0;
    let Inst{31-24} = Opcode;
    let OutOperandList = outs;
    let InOperandList = ins;
    let AsmString = asmstr;
    let Pattern = pattern;
    let Itinerary = itin;
    bits<4> FormBits = Form.Value;
    let TSFlags{3-0} = FormBits;
    let DecoderNamespace = "Cpu0";
    field bits<32> SoftFail = 0;
}
```

Figura 4.10: Implementação da classe de instruções genérica dentro do escopo da `TableGen`.

Para as instruções do formato A, a classe `FA` foi implementada. Os operandos **ra**, **rb**, **rc** e o **shamt** são definidos a partir dos bits da instrução, conforme Figura 4.11.

```
//-----//
// Format A instruction class in Cpu0 : <|opcode|ra|rb|rc|cx|>
//-----//
class FA<bits<8> op, dag outs, dag ins, string asmstr,
        list<dag> pattern, InstrItinClass itin>:
    Cpu0Inst<outs, ins, asmstr, pattern, itin, FmFA>
{
    bits<4> ra;
    bits<4> rb;
    bits<4> rc;
    bits<12> shamt;

    let Opcode = op;

    let Inst{23-20} = ra;
    let Inst{19-16} = rb;
    let Inst{15-12} = rc;
    let Inst{11-0} = shamt;
}
```

Figura 4.11: Implementação da classe de formato de instruções do tipo A dentro do escopo da `TableGen`.

De semelhante modo, as classes `FL` e `FJ` foram implementadas para as instruções de formato

L e J, respectivamente, conforme suas particularidades apresentadas no Capítulo 3:

```
//=====//
// Format L instruction class in Cpu0 : <|opcode|ra|rb|cx|>
//=====//

class FL<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
    InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>
{
    bits<4> ra;
    bits<4> rb;
    bits<16> imm16;

    let Opcode = op;

    let Inst(23-20) = ra;
    let Inst(19-16) = rb;
    let Inst(15-0) = imm16;
}
```

Figura 4.12: Implementação da classe de formato de instruções do tipo L dentro do escopo da TableGen.

```
//=====//
// Format J instruction class in Cpu0 : <|opcode|address|>
//=====//

class FJ<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
    InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmJ>
{
    bits<24> addr;

    let Opcode = op;

    let Inst(23-0) = addr;
}
```

Figura 4.13: Implementação da classe de formato de instruções do tipo J dentro do escopo da TableGen.

4.2.4.2 Descrição do conjunto de instruções

A partir das descrições de formatos fornecidas anteriormente, as instruções podem ser descritas detalhadamente. Para fins de exemplo, iremos demonstrar a definição das instruções ADD e ORi para explicar o processo de descrição de uma instrução, que é realizado para cada uma das instruções do processador. Essas operações realizam, respectivamente:

1. $Ra \leftarrow Rb + Rc$
2. $Ra \leftarrow Rb \text{ OR } Cx$

As Figuras 4.14 e 4.15 ilustram seus formatos, respectivamente:

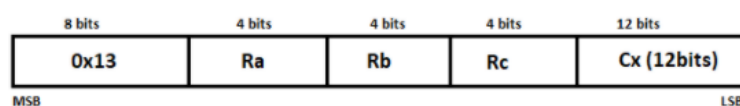


Figura 4.14: Formato da instrução ADD.

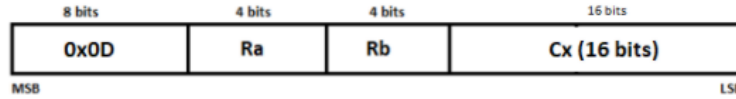


Figura 4.15: Formato da instrução ORi.

Primeiramente, é preciso descrever os operandos das instruções. Muitos operandos já estão pre-definidos na TableGen, no entanto, há situações onde isso não é suficiente. No caso do ADD, os operandos Ra, Rb e Rc são todos registradores e são descritos em CpuRegisterInfo.td (conforme a seção 4.4), sendo assim, pode-se definir diretamente a instrução:

```
def ADD      : ArithLogicR<0x13, "add", add, IIALu, CPURegs, 1>;
```

Figura 4.16: Definição da instrução ADD dentro do escopo da TableGen.

Os parâmetros passados são o **opcode (0x13)**, a **string da instrução ("add")**, o **opNode (add)**, que é o tipo de SDNode usado como *pattern* no momento de seleção da instrução, o **tipo de itinerário (IIALu)**, que será usado no escalonamento de instruções, a **classe de registradores dos operandos (CPURegs)** e a **flag isCommutable (1)**, que define se a operação em questão é ou não comutativa. Nota-se que a ADD é definida a partir da classe ArithLogicR. Tal classe foi implementada a partir da classe FA (apresentada na seção anterior) e agrega informações comuns às operações lógico-aritméticas cujos operandos sejam somente registradores. Os operandos de entrada e saída assumem o formato "local:\$nome", onde 'local' é uma classe de registradores ou uma instância (pre-definida ou customizada) de *Operand* (classe interna da TableGen) e \$nome é usado pelo *AsmPrinter* no processo de emissão de código. Além disso, a linha 5 da Figura 4.17 descreve o *pattern* que permite o gerador de código selecionar a instrução de máquina corretamente a partir do código LLVM IR.

```
1 class ArithLogicR<bits<8> op, string instr_asm, SDNode OpNode,
2     InstrItinClass itin, RegisterClass RC, bit isComm = 0>:
3     FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
4     !strconcat(instr_asm, "\t$ra, $rb, $rc"),
5     [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], itin> {
6     let shamt = 0;
7     let isCommutable = isComm;    // e.g. add rb rc = add rc rb
8     let isReMaterializable = 1;
9 }
```

Figura 4.17: Implementação da classe ArithLogicR, classe utilizada para definir as instruções lógicas e aritméticas cujos operandos são somente registradores.

No caso do ORi, define-se o operando imediato da forma a seguir (isso é feito para qualquer instrução que utilize operandos especiais, como por exemplo as instruções de saltos, que possuem endereço de destino com tamanho de 24 bits.). O imediato é definido como uma instância da classe interna da TableGen *Operand*, onde informa-se à TableGen o nome do método da classe *AsmPrinter* responsável pela sua emissão em assembly (na fase de emissão de código).

Assim, define-se a instrução:

```
def uimm16      : Operand<i32> {
  let PrintMethod = "printUnsignedImm";
}
```

Figura 4.18: Definição do operando imediato utilizado pela instrução ORi.

```
def ORi      : ArithLogicI<0x0d, "ori", or, uimm16, immZExt16, CPURegs>;
```

Figura 4.19: Definição da instrução ORi dentro do escopo da TableGen.

Nesse caso, uma nova classe base é utilizada, a `ArithLogicI`, que é definida a partir da classe `FL` (uma vez que a instrução ORi é do formato L):

```
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
  Operand Od, PatLeaf imm_type, RegisterClass RC> :
  FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
  !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
  [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIALu> {
  let isReMaterializable = 1;
}
```

Figura 4.20: Implementação da classe `ArithLogicI`, classe utilizada para definir as instruções lógicas e aritméticas cujos operandos são registradores e valores imediatos.

4.2.4.3 O seletor de instruções

Uma vez que todas as instruções foram definidas, é possível realizar efetivamente a seleção de instrução de máquina para cada um dos nós do DAG legalizado, transformando-o em um DAG de instruções da máquina alvo. Isso é feito por meio da correspondência entre os nós do DAG e os respectivos *patterns* das instruções, conforme a sequência de etapas a seguir:

- 1) O gerador de código executa o método `SelectionDagISel::DoInstructionSelection` que é responsável por invocar o seletor de instruções conforme o target utilizado. No caso, o método `Cpu0DagToDagISel::Select` é invocado para analisar e executar a seleção de instrução para cada nó do DAG legalizado.

- 2) Caso houvesse correspondências de instruções a serem feitas de forma manual, o método `Cpu0ISelDagToDag::Select` as realizaria e, posteriormente, invocaria o método responsável pelas correspondências automáticas. No caso do `Cpu0`, nenhuma correspondência manual é necessária, de modo que o método `CpuDagToDagISel::SelectCode` é invocado para realizar as correspondências automáticas. Este método é gerado automaticamente a partir dos arquivos de descrição TableGen das instruções, descritos acima.

- 3) O `Cpu0ISelDagToDag::SelectCode` faz, para cada um dos nós do DAG em transformação, uma varredura entre as instruções buscando a correspondência correta. Pode ser que haja mais de uma instrução cujo *pattern* corresponda adequadamente ao nó. Nesses casos, a primeira correspondência encontrada é utilizada.

4.2.4.4 Informações não estáticas

Além das descrições fornecidas a partir da ferramenta TableGen, é necessário implementar alguns métodos que analisam, inserem e transformam algumas instruções. Esses métodos são convocados na etapa final da geração de código e realizam, por exemplo, análises de saltos (inserindo-os ou removendo-os, conforme o caso), manipulações de registradores para uma utilização eficiente dos mesmos, etc.

Assim, o DAG final da etapa de seleção de instruções é mostrado na Figura 4.21, transformado a partir do DAG da Figura 4.9:

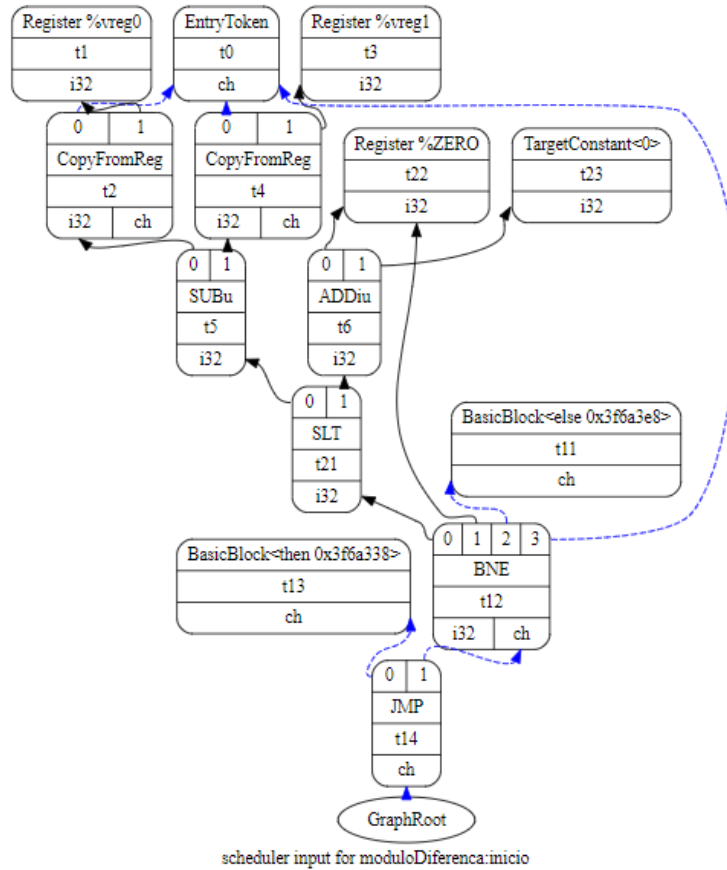


Figura 4.21: DAG final do bloco "inicio" da função moduloDiferenca.

4.3 Escalonamento de Instruções

O escalonamento de instruções é a fase em que o DAG legalizado é transformado em uma lista de instruções, que são representadas por instâncias da classe MachineInstr. Essa transformação é feita buscando minimizar a latência do processador e otimizar o uso dos registradores. O LLVM faz uso de informações como quais são as unidades funcionais do processador, quais delas são utilizadas por cada instrução, suas latências, etc., para se ter um bom escalonamento. Após a geração da lista, o DAG é destruído. No caso do Cpu0, o desenvolvedor do backend decidiu

não fornecer muitos detalhes desses aspectos da arquitetura. A lista de instruções geradas pelo escalonador a partir do DAG da Figura 4.21 é:

```
BB#0: derived from LLVM BB %inicio
Live Ins: %A0 %A1
%vreg1<def> = COPY %A1; CPURegs:%vreg1
%vreg0<def> = COPY %A0; CPURegs:%vreg0
%vreg2<def> = SUBu %vreg0, %vreg1; GPROut:%vreg2 CPURegs:%vreg0,%vreg1
%vreg3<def> = ADDiu %ZERO, 0; GPROut:%vreg3
%vreg4<def> = SLT %vreg2, %vreg3; GPROut:%vreg4,%vreg2,%vreg3
BNE %vreg4, %ZERO, <BB#2>, %AT<imp-def,dead>; GPROut:%vreg4
JMP <BB#1>
```

Figura 4.22: Código gerado a partir do bloco "inicio" da função moduloDiferenca após etapa "Escalonamento de Instruções".

4.4 Alocação dos registradores

Nesta etapa, o gerador de código realiza a alocação dos registradores. Para isso, é necessário descrever detalhadamente o banco de registradores ao LLVM. A maior parte das informações são fornecidas por meio da TableGen. No entanto, como será visto, algumas informações dependem de fatores não conhecidos antes da execução, e precisam ser implementadas separadamente em códigos C++.

4.4.0.1 Descrição do banco de registradores

A partir da classe Register fornecida pela TableGen foram implementadas as classes de registradores do Cpu0.

```
class Cpu0Reg<bits<16> Enc, string n> : Register<n> {
    let HWEncoding = Enc;
    let Namespace = "Cpu0";
}

class Cpu0GPRReg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;

class Cpu0C0Reg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;
```

Figura 4.23: Implementação da classe Cpu0Reg utilizada para definir os registradores do Cpu0 dentro do escopo da TableGen.

A classe Cpu0Reg define genericamente os registradores do Cpu0. A classe Cpu0GPRReg foi implementada para definir os registradores de uso geral e a classe Cpu0C0Reg define os registradores de uso específico ao processador, conforme a Figura 4.24. Cada registrador deve ser definido como uma instância de uma das classes acima. O número identificador e o nome do registrador são passados como argumentos. Ainda, o campo DWARF é definido para fins de depuração.

```

def ZERO : Cpu0GPRReg<0, "zero">, DwarfRegNum<[0]>;
def AT   : Cpu0GPRReg<1, "1">, DwarfRegNum<[1]>;
def V0   : Cpu0GPRReg<2, "2">, DwarfRegNum<[2]>;
def V1   : Cpu0GPRReg<3, "3">, DwarfRegNum<[3]>;
def A0   : Cpu0GPRReg<4, "4">, DwarfRegNum<[4]>;
def A1   : Cpu0GPRReg<5, "5">, DwarfRegNum<[5]>;
def T9   : Cpu0GPRReg<6, "t9">, DwarfRegNum<[6]>;
def T0   : Cpu0GPRReg<7, "7">, DwarfRegNum<[7]>;
def T1   : Cpu0GPRReg<8, "8">, DwarfRegNum<[8]>;
def S0   : Cpu0GPRReg<9, "9">, DwarfRegNum<[9]>;
def S1   : Cpu0GPRReg<10, "10">, DwarfRegNum<[10]>;
def GP   : Cpu0GPRReg<11, "gp">, DwarfRegNum<[11]>;
def FP   : Cpu0GPRReg<12, "fp">, DwarfRegNum<[12]>;
def SP   : Cpu0GPRReg<13, "sp">, DwarfRegNum<[13]>;
def LR   : Cpu0GPRReg<14, "lr">, DwarfRegNum<[14]>;
def SW   : Cpu0GPRReg<15, "sw">, DwarfRegNum<[15]>;
def HI   : Cpu0Reg<0, "ac0">, DwarfRegNum<[18]>;
def LO   : Cpu0Reg<0, "ac0">, DwarfRegNum<[19]>;
def PC   : Cpu0C0Reg<0, "pc">, DwarfRegNum<[20]>;
def EPC  : Cpu0C0Reg<1, "epc">, DwarfRegNum<[21]>;

```

Figura 4.24: Definição do banco de registradores do Cpu0 dentro do escopo da TableGen.

Por fim, o LLVM estabelece a necessidade de definir uma ou mais classes de registradores por meio da classe interna `RegisterClass`. Essas classes são utilizadas pelo gerador de código no processo de seleção de instruções, uma vez que toda instrução que possuir um ou mais registradores como operandos deve especificar a quais classes de registradores esses operandos pertencem. Cada `RegisterClass` é então especificada por: o tipo de dado, alinhamento e um conjunto de registradores que lhe pertencem. Vale salientar que um registrador pode pertencer a uma ou mais classes de registradores. As classes de registradores do Cpu0 foram definidas conforme a Figura 4.25.

```

def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add
  // Reserved
  ZERO, AT,
  // Return Values and Arguments
  V0, V1, A0, A1,
  // Not preserved across procedure calls
  T9, T0, T1,
  // Callee save
  S0, S1,
  // Reserved
  GP, FP,
  SP, LR, SW)>;

//@Status Registers class
def SR : RegisterClass<"Cpu0", [i32], 32, (add SW)>;

//@Co-processor 0 Registers class
def C0Regs : RegisterClass<"Cpu0", [i32], 32, (add PC, EPC)>;

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPURegs, SW))>;

```

Figura 4.25: Definição das classes de registradores do Cpu0 dentro do escopo da TableGen.

4.4.0.2 Informações não-estática

Conforme mencionado anteriormente, algumas características dos registradores não podem ser definidas estaticamente por meio da TableGgen, mas sim determinadas durante a execução. Essas informações estão implementadas na classe Cpu0RegisterInfo. Entre elas, temos:

1) Registradores reservados: alguns dos registradores são de uso específico - como por exemplo R0 contém sempre o valor zero e LR guarda o endereço de retorno após desvios - e por isso não podem estar disponíveis para uso geral na etapa de alocação de registradores. O Cpu0RegisterInfo.cpp implementa um método para marcar todos os registradores de uso reservado num vetor de bits.

2) Registradores callee-saved: normalmente a ABI define um conjunto de registradores que devem ser preservados durante a chamada e retorno de uma função (chamados de registradores callee-saved). No caso do Cpu0, que se baseia na ABI do Mips [8], os registradores LR e FP foram definidos como registradores callee-saved.

3) Frame Register: o *frame register* (FP) é um endereço base a partir do qual pode-se acessar à pilha (memória). Normalmente, o tamanho do frame é fixo e por essa razão o endereço da pilha pode ser obtido por meio do *stack pointer* (SP). Caso o tamanho do frame seja variável, faz-se necessário utilizar o FP.

Além dessas informações, o Cpu0RegisterInfo contém alguns métodos que emitem partes do código. Esses métodos são invocados pelo gerador de código na etapa de “emissão do prólogo e epílogo”, quando a seleção de instruções e alocação dos registradores foram realizadas e o código LLVM IR encontra-se traduzido para linguagem da máquina alvo. Entre eles, temos:

1) EliminateFrameIndex(): esta função é sempre invocada quando o código gerado contém alguma instrução de acesso à memória. Antes de ser chamada, o gerador de código faz referência à memória por meio de um índice de frame abstrato e de um imediato. Após a sua execução, as referências à memória são substituídas por um registrador acrescentado de um valor de offset real. Caso a função possua um frame fixo ou variável, esse registrador pode ser tanto o *frame pointer* (FP) quanto o *stack pointer* (SP), respectivamente, e o *offset* é calculado de acordo com o registrador utilizado.

2) EliminateCallFramePseudoInstr(): Todas as vezes que uma instrução do tipo **call** é emitida, este método insere, respectivamente, as pseudo-instruções ADJCALLSTACKDOWN e ADJCALLSTACKUP antes e depois daquela instrução, de modo que:

- se a função que estiver sendo chamada tiver um *stack frame* fixo, essas pseudo-instruções são removidas, já que o espaço para seus argumentos já foi alocado no prólogo da função
- se o tamanho de seu *stack frame* for variável, estas funções são substituídas por adições ou subtrações ao *stack pointer* de modo a acessar o *stack frame* corretamente.

Após a alocação de registradores, o código para o bloco "inicio" da função da Figura 4.22 é:


```

BB#0: derived from LLVM BB %inicio
Live Ins: %A0 %A1
%V0<def> = SUBu %A0, %A1
%V1<def> = ADDiu %ZERO, 0
%V1<def> = SLT %V0, %V1<kill>
BNE %V1<kill>, %ZERO, <BB#2>, %AT<imp-def,dead>
JMP <BB#1>

```

Figura 4.26: Código gerado a partir do bloco "inicio" da função moduloDiferenca após a etapa "Alocação de registradores".

4.5 Inserção do prólogo e epílogo

Nesta fase, o prólogo e o epílogo da função são inseridos. Isso é feito pelos métodos `EmitPrologue()` e `EmitEpilogue()` da classe `Cpu0SEFrameLowering`:

`EmitPrologue()`: Esse método, ao ser executado, insere o prólogo no início das funções. No prólogo da função é realizado a reserva de espaço para o quadro de pilha da função. Para isso basta adicionar ao SP o tamanho do quadro. Se for necessária a utilização do FP, o mesmo toma o valor de SP.

`EmitEpilogue()`: esse método insere o epílogo da função e realiza o processo inverso do `EmitPrologue()`, uma vez que libera o espaço reservado para o quadro da pilha e restaura os valores dos registradores SP e FP.

Além desses, um outro método também é chamado nesta fase:

`spillCalleeSavedRegisters()`: Esse método realiza a restauração dos registradores *callee-saved*. No caso do Cpu0 os únicos registradores definidos como *callee saved* foram o LR e FP.

Após a inserção do prólogo e epílogo, o código para o bloco "inicio" da função da Figura 4.4 é:

```

BB#0: derived from LLVM BB %inicio
Live Ins: %A0 %A1
%V0<def> = SUBu %A0, %A1
%V1<def> = ADDiu %ZERO, 0
%V1<def> = SLT %V0, %V1<kill>
BNE %V1<kill>, %ZERO, <BB#2>, %AT<imp-def,dead>
JMP <BB#1>

```

Figura 4.27: Código gerado a partir do bloco "inicio" da função moduloDiferenca após a etapa "Inserção do prólogo e epílogo".

4.6 Emissão do código

A fase final do processo de geração de código é a emissão do código em formato *assembly*. Essa tarefa é executada pelo chamado *assembly printer* (implementado na classe `CpuAsmPrinter`). O processo é efetuado de forma direta: para cada uma das funções existentes o *assembly printer* invoca o método `RunOnMachineFunction()` para imprimir o *header* da função e em seguida processar

os diferentes blocos da função (representados pela classe `MachineBasicBlock`). Para cada instrução do bloco em tratamento, o método `printInstruction()` é invocado para emití-las no formato *assembly*. Não obstante, para cada operando da instrução em análise, o método `printOperand()` é invocado para emití-lo, com exceção dos operandos de memória (do tipo *reigstrador/imediato*), que são emitidos pelo método `printMemOperand()`. Esses métodos de emissão de códigos, em grande parte, são gerados automaticamente pela `TableGen` a partir dos arquivos de descrição do backend. Por fim, o código final em formato *assembly* para o código IR da função da Figura 4.4 é:

```
# BB#0:                                     # %inicio
    subu    $2, $4, $5
    addiu   $3, $zero, 0
    slt     $3, $2, $3
    bne     $3, $zero, $BB0_2
    nop
# BB#1:                                     # %then
    ret     $lr
    nop
$BB0_2:                                     # %else
    subu    $2, $5, $4
    ret     $lr
    nop
```

Figura 4.28: Código final em assembly gerado a partir da função `moduloDiferenca`.

4.7 Integração ao LLVM

O último passo no desenvolvimento do backend é integrá-lo à estrutura do LLVM. Isso é feito por meio da alteração de alguns arquivos do sistema LLVM que possibilitam o `Cpu0` backend ser reconhecido e utilizado. O LLVM possui arquivos de configuração para os seguintes sistemas de construção: GNU Autotools e Cmake. Sendo assim, o diretório do `Cpu0` foi inserido nos arquivos de configuração de ambos sistemas a fim de que o mesmo possa ser compilado junto aos outros componentes do LLVM. Além disso, para que o LLVM possa reconhecer o `Cpu0`, foi preciso registrá-lo por meio da classe `TargetRegistry` que viabiliza o acesso e a utilização do backend.

```
extern "C" void LLVMInitializeCpu0TargetInfo() {
    RegisterTarget<Triple::cpu0,true> X(TheCpu0Target, "cpu0", "Cpu0");
}
```

Figura 4.29: Método de registro do backend `Cpu0` ao LLVM.

O LLVM estabelece uma *string target triple* usada para identificar os diferentes processadores. A *string* é composta por três partes: arquitetura, fabricante e sistema operacional, conforme a Figura 4.29. A partir disso, o LLVM está apto a encontrar o `Cpu0` por meio da correspondência dessa *string*.

Capítulo 5

Modificações, Testes e Resultados

Neste capítulo apresenta-se algumas modificações realizadas no backend LLVM do Cpu0, bem como os resultados de uma seleção de testes e simulações realizados para verificar a geração de código pelo backend.

5.1 Verificação do backend no simulador de Verilog

Nesta seção apresenta-se um conjunto de testes que buscam verificar o backend Cpu0. Para cada teste mostra-se o resultado obtido da compilação (código gerado pelo backend a partir do código de entrada em LLVM IR) e o resultado da simulação da respectiva execução por meio do simulador de Verilog chamado "Icarus Verilog"[17]. Ainda, os códigos dos testes foram escritos inicialmente em C e transformados em LLVM IR a partir do frontend Clang (vide seção 2.3.1).

O primeiro teste realizado foi o teste da função da Figura 5.1, que simplesmente realiza a soma entre dois números e retorna o resultado. O respectivo código em LLVM IR encontra-se na Figura 5.2.

```
int main()
{
    int a, b, c;
    a = 5;
    b = 3;
    c = a+b;
    return c;
}
```

Figura 5.1: Código em C da função que soma dois números quaisquer.

```

define i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 5, i32* %2, align 4
    store i32 3, i32* %3, align 4
    %5 = load i32, i32* %2, align 4
    %6 = load i32, i32* %3, align 4
    %7 = add nsw i32 %5, %6
    store i32 %7, i32* %4, align 4
    %8 = load i32, i32* %4, align 4
    ret i32 %8
}

```

Figura 5.2: Código em LLVM IR da função que soma dois números quaisquer.

Utilizando o backend Cpu0 obteve-se o código em *assembly* da Figura 5.3:

```

# BB#0:
    addiu    $sp, $sp, -24
    st      $fp, 20($sp)
    move     $fp, $sp
    addiu    $2, $zero, 0
    st      $2, 16($fp)
    addiu    $2, $zero, 5
    st      $2, 12($fp)
    addiu    $2, $zero, 3
    st      $2, 8($fp)
    ld      $2, 12($fp)
    addiu    $2, $2, 3
    st      $2, 4($fp)
    move     $sp, $fp
    ld      $fp, 20($sp)
    addiu    $sp, $sp, 24
    ret      $lr
    nop

```

Figura 5.3: Código em *assembly* da função que soma dois números quaisquer.

A partir disso, pode-se simular a execução do código conforme a Figura 5.4:

clock =																			
ir[31:0] =	09DD+	02CD+	11CD+	0920+	022C+	0920+	022C+	0920+	022C+	012C+	0922+	022C+	11DC+	01CD+	09DD+	3CE0+	00+		
op[7:0] =	9	2	17	9	2	9	2	9	2	1	9	2	17	1	9	60	0		
mar[31:0] =	000+	+	00+	0000+	0000+	+	00+	0000+	+	00+	0000+	+	00+	0000+	+	00+	0000+	0000+	000+
mdr[31:0] =	1655+	0	2986+	1530+	0	1530+	5	1530+	3	19+	5	1532+	8	2996+	30+	0	1654+	1021+	0
data[31:0] =	458724		0		5		3		8		458724		458748						
i[3:0] =	13		12		2								13						

Figura 5.4: Resultado obtido da simulação da execução do código da Figura 5.3

A simulação retrata o opcode das instruções executadas. Em laranja, tem-se o valores dos registradores MAR e MDR referentes à memória. As duas linhas mais abaixo representam o valor e qual registrador é afetado durante a execução. Pode-se perceber que os valores das variáveis são armazenados em memória e que o resultado da soma é correto.

O segundo teste realizado teve o objetivo de verificar o funcionamento da estrutura de decisão if-else. A Figura 5.5 retrata a função implementada para isso:

```
int main()
{
    int a, b;
    a = 60;
    b = -1;
    if(a > 0){
        b = 1;
    }
    else{
        b = 0;
    }
}
```

Figura 5.5: Código em C da função que testa a estrutura if-else.

Seu respectivo código em LLVM IR gerado pelo Cang é ilustrada na Figura 5.6:

```
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 60, i32* %2, align 4
    store i32 -1, i32* %3, align 4
    %4 = load i32, i32* %2, align 4
    %5 = icmp sgt i32 %4, 0
    br i1 %5, label %6, label %7

; <label>:6:
    store i32 1, i32* %3, align 4
    br label %8

; <label>:7:
    store i32 0, i32* %3, align 4
    br label %8

; <label>:8:
    %9 = load i32, i32* %1, align 4
    ret i32 %9
}
```

Figura 5.6: Código em LLVM IR da função que testa a estrutura if-else.

De modo semelhante ao primeiro teste, utilizou-se o backend Cpu0 para gerar o código em *assembly* da Figura 5.7:

```

# BB#0:
    addiu    $sp, $sp, -16
    st      $fp, 12($sp)
    move     $fp, $sp
    addiu    $2, $zero, 0
    st      $2, 8($fp)
    addiu    $3, $zero, 60
    st      $3, 4($fp)
    addiu    $3, $zero, -1
    st      $3, 0($fp)
    ld      $4, 4($fp)
    addiu    $3, $zero, 1
    slt      $4, $4, $3
    bne      $4, $zero, $BB0_2
    nop

# BB#1:
    st      $3, 0($fp)
    jmp      $BB0_3
$BB0_2:
    st      $2, 0($fp)
$BB0_3:
    ld      $2, 8($fp)
    move     $sp, $fp
    ld      $fp, 12($sp)
    addiu    $sp, $sp, 16
    ret      $lr
    nop

```

Figura 5.7: Código em *assembly* da função que testa a estrutura if-else.

O resultado da simulação da execução do código é:

clock =																																				
ir[31:0] =	09D+	02C+	11C+	092+	022+	093+	023+	093+	023+	014+	093+	10F+	320+	000+	023+	360+	012+	11D+	01C+	09D+	3CE+	0+														
op[7:0] =	9	2	17	9	2	9	2	9	2	1	9	16	50	0	2	54	1	17	1	9	60	0														
mar[31:0] =	00+	+	0+	000+	000+	+	0+	000+	+	0+	000+	000+	000+	000+	+	0+	000+	+	0+	000+	000+	00+														
mdr[31:0] =	165+	0	298+	153+	0	154+	60	154+	42+	2+	+	154+	284+	838+	0	1	905+	1+	0	299+	3+	0	165+	102+	0											
data[31:0] =	458732	0				60	-1				1												458732	458748												
i[3:0] =	13	12	2	3																				13												

Figura 5.8: Resultado obtido da simulação da execução do código da Figura 5.5

Essa simulação permite visualizar os mesmos dados que a simulação do teste anterior. É possível notar que a execução ocorreu corretamente, uma vez que o salto condicional não foi tomado (o que era esperado) e que o valor do registrador \$3 é 1.

Por fim, o último teste feito visou verificar a chamada de função. A Figura 5.9 retrata esta

configuração.

```
int main()
{
    int a, b, c;
    a = 5;
    b = 3;
    c = funcSoma(a,b);
    return 0;
}

int funcSoma(int n1, int n2)
{
    return n1+n2;
}
```

Figura 5.9: Código em C que testa a chamada de função.

As Figuras 5.10 e 5.11 retratam os códigos em LLVM IR para cada uma das funções:

```
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 3, i32* %2, align 4
    store i32 5, i32* %3, align 4
    %5 = load i32, i32* %2, align 4
    %6 = load i32, i32* %3, align 4
    %7 = call i32 @Z8funcSomaii(i32 signext %5, i32 signext %6)
    store i32 %7, i32* %4, align 4
    ret i32 0
}
```

Figura 5.10: Código em LLVM IR da função main.

```
define i32 @_Z8funcSomaii(i32 signext, i32 signext) #1 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, i32* %3, align 4
    store i32 %1, i32* %4, align 4
    %5 = load i32, i32* %3, align 4
    %6 = load i32, i32* %4, align 4
    %7 = add nsw i32 %5, %6
    ret i32 %7
}
```

Figura 5.11: Código em LLVM IR da função funcSoma.

Da mesma forma, as Figuras 5.12 e 5.13 retratam os respectivos códigos das funções em *assembly*, gerados pelo backend Cpu0:

```

# BB#0:
    addiu    $sp, $sp, -40
    st      $lr, 36($sp)
    st      $fp, 32($sp)
    st      $9, 28($sp)
    move     $fp, $sp
    st      $9, 24($fp)
    addiu    $9, $zero, 0
    addiu    $2, $zero, 5
    st      $2, 20($fp)
    addiu    $5, $zero, 3
    st      $5, 16($fp)
    ld      $4, 20($fp)
    jsub     40
    nop
    st      $2, 12($fp)
    addu     $2, $zero, $9
    move     $sp, $fp
    ld      $9, 28($sp)
    ld      $fp, 32($sp)
    ld      $lr, 36($sp)
    addiu    $sp, $sp, 40
    ret      $lr
    nop

```

Figura 5.12: Código em *assembly* da função main.

```

_Z8funcSomaii:
    addiu    $sp, $sp, -16
    st      $fp, 12($sp)
    move     $fp, $sp
    st      $4, 8($fp)
    st      $5, 4($fp)
    ld      $2, 8($fp)
    addu     $2, $2, $5
    move     $sp, $fp
    ld      $fp, 12($sp)
    addiu    $sp, $sp, 16
    ret      $lr
    nop

```

Figura 5.13: Código em *assembly* da função funcSoma.

O resultado da simulação é ilustrado pelas Figuras 5.14 e 5.15:

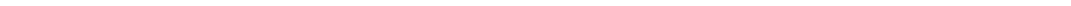
clock =																																
pc[31:0] =	96	100	104	108	112	116	120	124	128	132	136	140	144	184	188	192	196	200	204													
ir[31:0] =	09DD+	02ED0+	02CD0+	029D0+	11CD0+	09900+	029C0+	09200+	022C0+	09500+	025C0+	014C0+	38000+	00000+	09DDF+	02CD0+	11CD0+	024C0+	02													
op[7:0] =	9	2				17	9	2	9	2	9	2	1	59	0	9	2	17	2													
mar[31:0] =	000+	0+	00+	0+	00+	0+	00+	00000+	00000+	0+	00+	00000+	0+	00+	00000+	0+	00+	00000+	00000+	00000+	0+	00+	00000+	0+	00+	00000+	00000+	00000+	00000+	00000+	00000+	
mdr[31:0] =	16554+	4294+	0	0	0	29864+	16043+	0	15309+	3	15623+	5	21+	3	98985+	0	16554+	4587+	29864+	3												
data[31:0] =	458708					0					3					5													458692			
i[3:0] =	13					12					9					2					5									13	12	

Figura 5.14: Resultado obtido da simulação da execução do código da Figura 5.9 (1 de 2)

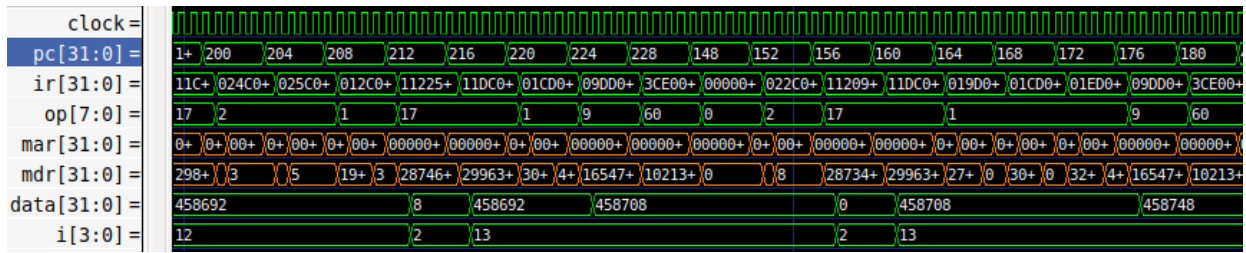


Figura 5.15: Resultado obtido da simulação da execução do código da Figura 5.9 (2 de 2)

Além dos dados mostrados nas outras simulações, nesta simulação apresenta-se o valor do registrador PC. Por meio dele pode-se notar que a chamada da função ocorre ao saltar de 144 para 184 (conforme a instrução em *assembly* **jsub 40**). Da Figura 5.15 percebe-se o retorno da função quando o valor de PC muda de 228 para 148. Além disso, percebe-se dos valores dos registradores e dos valores guardados em memória que o programa executou corretamente.

5.2 Omitindo uma instrução do backend

Nesta seção é discutido os passos necessários para informar ao backend, por exemplo, que o Cpu0 foi alterado de modo a remover de sua ISA uma instrução. No caso em questão, as instruções que deseja-se remover são as instruções de multiplicação. Busca-se, assim, fazer o backend gerar automaticamente um código equivalente para tais instruções.

No primeiro momento, optou-se por simplesmente retirar do arquivo Cpu0InstrInfo.td a definição de tais instruções e observar como o backend se comporta. A partir disso, ao tentar compilar o código em LLVM IR da Figura 5.16, obtêve-se a mensagem de erro da Figura 5.17:

```
define i32 @_mulTest(i32 signext, i32 signext) #1 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, i32* %3, align 4
    store i32 %1, i32* %4, align 4
    %5 = load i32, i32* %3, align 4
    %6 = load i32, i32* %4, align 4
    %7 = mul nsw i32 %5, %6
    ret i32 %7
}
```

Figura 5.16: Código LLVM IR utilizado para verificar a omissão de instruções de multiplicação.

LLVM ERROR: Cannot select: t13: i32 = mul t11, t4

Figura 5.17: Mensagem de erro obtido ao omitir as definições de instruções de multiplicação no backend Cpu0.

Não obstante, ao inserir, no escopo da classe Cpu0TargetLowering, os comandos (Figura 5.18),

que dizem ao gerador de código para expandir as instruções de multiplicação, foi possível obter o código em assembly da Figura 5.19:

```
setOperationAction(ISD::MUL,          MVT::i32, Expand);
setOperationAction(ISD::MULHS,       MVT::i32, Expand);
setOperationAction(ISD::MULHU,       MVT::i32, Expand);
setOperationAction(ISD::SMUL_LOHI,   MVT::i32, Expand);
setOperationAction(ISD::UMUL_LOHI,   MVT::i32, Expand);
```

Figura 5.18: Comandos *SetOperationAction* para expandir instruções de multiplicação.

```
# BB#0:
    lui $2, %hi(_gp_disp)
    addiu $2, $2, %lo(_gp_disp)
    addiu $sp, $sp, -24
    st $lr, 20($sp)           # 4-byte Folded Spill
$tmp1:
    st $4, 16($sp)
    st $5, 12($sp)
    ld $t9, %call16(__mulsi3)($gp)
    ld $4, 16($sp)
    jalr $t9
    nop
    ld $gp, 8($sp)
    ld $lr, 20($sp)          # 4-byte Folded Reload
    addiu $sp, $sp, 24
    ret $lr
    nop
```

Figura 5.19: Código em *assembly* da função da Figura 5.16

Nota-se que o gerador de código invoca a função *mulsi3*, que é utilizada para realizar operações equivalentes à multiplicação, conforme Figura 5.20:

```

unsigned int
__mulsi3 (unsigned int a, unsigned int b)
{
    unsigned int r = 0;

    while (a)
    {
        if (a & 1)
            r += b;
        a >>= 1;
        b <<= 1;
    }
    return r;
}

```

Figura 5.20: Código em C da função *mulsi3*.

Uma outra forma de tentar resolver tal problema seria implementar a transformação de instruções de multiplicação em *loops* de adições, quer seja codificando diretamente em C++ quer seja a partir da ferramenta TableGen, por meio de sua classe interna *Pat*, com a qual defini-se transformações *DAG-to-DAG*. No entanto, o autor não teve tempo hábil para implementar tais abordagens.

5.3 Inserindo uma instrução no backend

Nesta seção será apresentado as implementações realizadas para incorporar ao backend Cpu0 um nova instrução, a saber, a instrução MAC (*Multiply-And-Accumulate*). Essa instrução computa o produto entre dois números e adiciona este produto a um acumulador, armazenando o resultado da operação neste mesmo acumulador:

$$a \leftarrow a + (b \times c)$$

Figura 5.21: Operação da instrução MAC.

Conforme mostrado no Capítulo 4, para incorporar a instrução ao backend é necessário, no mínimo, descrever tal instrução para a ferramenta TableGen, por meio de seus arquivos de descrição. Primeiramente, foi escrito a classe *MACFormat*, que foi implementada a partir da classe *Cpu0Inst*, conforme Figura 5.22:

```

1 class MACFormat:
2   Cpu0Inst<(outs GPROut:$ra), (ins CPURegs:$rb, CPURegs:$rc, CPURegs:$rd),
3   "mac $ra, $rb, $rc, $rd",
4   [(set GPROut:$ra, (add CPURegs:$rb, (mul CPURegs:$rc, CPURegs:$rd)))]], IIIMul, FrmA>
5 {
6   bits<4> ra;
7   bits<4> rb;
8   bits<4> rc;
9   bits<4> rd;
10
11   let Opcode = 0x53;
12
13   let Inst{23-20} = ra;
14   let Inst{19-16} = rb;
15   let Inst{15-12} = rc;
16   let Inst{12-9} = rd;
17 }

```

Figura 5.22: Implementação da classe MACFormat

O *opcode* atribuído foi 0x53, que é o próximo *opcode* disponível na ISA do Cpu0. Na linha 3 é especificada a *string* que será utilizado pelo *AsmPrinter* para emitir apropriadamente a instrução. Das linhas 6 a 16 é mostrado o mapeando dos operandos no decorrer dos bits da instrução. A linha 4 estabelece o *pattern* que será utilizado pelo seletor de instruções para selecionar a instrução.

Assim, pode-se definir a instrução, conforme Figura 5.23:

```
def MAC : MACFormat;
```

Figura 5.23: Definição da instrução MAC no arquivo Cpu0InstrInfo.

Após isso, ao testar a geração de código de máquina para o código da LLVM IR da Figura 5.24, obteve-se o código *assembly* da Figura 5.25. Pode-se perceber que a instrução MAC foi selecionada corretamente:

```

1 define i32 @_MACTest() #1 {
2   %1 = alloca i32, align 4
3   %2 = alloca i32, align 4
4   %3 = alloca i32, align 4
5   %4 = alloca i32, align 4
6   store i32 4, i32* %2, align 4
7   store i32 2, i32* %3, align 4
8   store i32 3, i32* %4, align 4
9   %5 = load i32, i32* %2, align 4
10  %6 = load i32, i32* %3, align 4
11  %7 = load i32, i32* %4, align 4
12  %8 = mul nsw i32 %6, %7
13  %9 = add nsw i32 %5, %8
14  store i32 %9, i32* %1, align 4
15  ret i32 0
16 }

```

Figura 5.24: Função MACTest em LLVM IR utilizada para verificar a implementação da instrução MAC.

```

1  # BB#0:
2      addiu    $sp, $sp, -16
3  $tmp0:
4      .cfi_def_cfa_offset 16
5      addiu    $2, $zero, 4
6      st      $2, 8($sp)
7      addiu    $2, $zero, 2
8      st      $2, 4($sp)
9      addiu    $2, $zero, 3
10     st      $2, 0($sp)
11     ld      $3, 4($sp)
12     ld      $4, 8($sp)
13     mac      $2, $4, $3, $2
14     st      $2, 12($sp)
15     addiu    $2, $zero, 0
16     addiu    $sp, $sp, 16
17     ret      $lr
18     nop

```

Figura 5.25: Código *assembly* gerado a partir da Figura 5.24.

Capítulo 6

Conclusões

O principal objetivo deste trabalho era estudar como realizar o desenvolvimento de um backend LLVM para uma arquitetura computacional não suportada anteriormente, de modo a portá-la de um compilador para linguagens de programação bastante disseminadas como C e C++. Neste trabalho buscou-se apresentar os principais passos para tal objetivo a partir do código do backend desenvolvido para um processador didático de código aberto chamado Cpu0.

O desenvolvimento do backend teve como base o *framework* de geração de código disponibilizado pelo LLVM. Este *framework* fornece um conjunto de algoritmos independentes do *target* que realiza as principais etapas da geração de código. Além disso, o gerador de código disponibiliza classes basilares a partir das quais muitas das partes do backend Cpu0 foram implementadas.

A ferramenta TableGen desempenha um papel de grande importância na construção do backend, uma vez que reduz consideravelmente o esforço em fornecer informações específicas da máquina alvo (no caso, o Cpu0) que são utilizados pelo *framework* de geração de código, como a descrição do conjunto de instruções e o banco de registradores.

Os testes e simulações realizados mostraram que a geração de código em *assembly* a partir do backend Cpu0 ocorreram corretamente. Ainda, foi possível realizar algumas pequenas modificações no backend a fim de adequá-lo a possíveis alterações realizáveis no processador para aplicá-lo em contextos específicos.

6.1 Perspectivas Futuras

Entre os possíveis encaminhamentos para este trabalho pode-se listar os seguintes:

- Alterações no código Verilog para suportar em hardware a instrução MAC.
- Evolução do backend de modo a fornecer informações detalhadas a respeito dos estágio de pipeline para o escalonador de instruções de modo a melhorar ainda mais o desempenho.
- Implementação do código Verilog em FPGA e testes e simulações do backend nesta plataforma.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] ASIP Designer. Disponível em: <<https://www.synopsys.com/dw/ipdir.php?ds=asip-designer>>.
- [2] TENSILICA Xtensa LX7 Processor. Disponível em: <https://ip.cadence.com/uploads/1099/TIP_PBXtensa.pdf>.
- [3] TENSILICA Customizable Processors. Disponível em: <<http://openasip.org/>>.
- [4] THE LLVM Target-Independent Code Generator. Disponível em: <<http://llvm.org/docs/CodeGenerator.html>>.
- [5] ZHONGCHENG, C. *CPU0 processor architecture*. Disponível em: <<http://ccckmit.wikidot.com/ocs:cpu0>>.
- [6] TENSILICA Customizable Processors. Disponível em: <<https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable>>.
- [7] CANIS, A. C. Legup: Open-source high-level synthesis research framework. *Department of Electrical and Computer Engineering, University of Toronto*, 2015.
- [8] CHUNG-SHU, C. *Creating an LLVM backend for the Cpu0 architecture*. Disponível em: <<http://jonathan2251.github.io/lbd/index.html>>.
- [9] LATNER, C. Lvm: An infrastructure for multi-stage optimization. *Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL*, dez. 2002.
- [10] LATNER, C. *The name of LLVM*. Disponível em: <<http://lists.llvm.org/pipermail/llvm-dev/2011-December/046445.html>>.
- [11] LLVM Language Reference Manual. Disponível em: <<http://llvm.org/docs/LangRef.html>>.
- [12] CLANG: a C language family frontend for LLVM. Disponível em: <<https://clang.llvm.org/>>.
- [13] CLANG vs Other Open Source Compilers. Disponível em: <<https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/comparison.html>>.
- [14] CLANG - Features and Goals. Disponível em: <<http://clang.llvm.org/features.html>>.
- [15] TABLEGEN Overview. Disponível em: <<https://llvm.org/docs/TableGen/index.html>>.

- [16] WRITING an LLVM Backend. Disponível em: <<https://llvm.org/docs/WritingAnLLVMBackend.html>>.
- [17] ICARUS Verilog. Disponível em: <<http://iverilog.icarus.com>>.